异步图书 www.epubit.com.cn

JavaScript 著名专家撰写
指导读者进入 JavaScript 框架设计的魔法指南

JavaScript 框架设计

（第 2 版）

司徒正美◎编著



中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

目　录

第10章 PC端的事件系统

写作
会议活动早知道
加入异步

版权信息

书名：JavaScript框架设计（第2版）

ISBN：978-7-115-46429-3

施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

- 编　　著　司徒正美

  责任编辑　张　涛

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

内容提要

本书全面讲解了JavaScript框架设计及相关的知识，主要内容包括种子模块、语言模块、浏览器嗅探与特征侦测、类工厂、选择器引擎、节点模块、数据缓存模块、样式模块、属性模块、PC端和移动端的事件系统、jQuery的事件系统、异步模型、数据交互模块、动画引擎、MVVM、前端模板（静态模板）、MVVM的动态模板、性能墙与复杂墙、组件、jQuery时代的组件方案、avalon2的组件方案、react的组件方案等。

本书适合前端设计人员、JavaScript开发者、移动UI设计者、程序员和项目经理阅读，也可作为相关专业学习用书和培训学校教材。

前言

距本书前一版出版已经3年了，这3年来前端技术也发生了很大变化。因为这3年中，出现了团购、P2P、O2O、直播等几大创业热潮，对前端的技术需求更强，也要求更高。许多公司为了迎合用户需求，也由PC端转移到移动端，全新的交互方式，及之前不曾遇到的性能问题，这些都需要全新的思路与框架来解决。旧的jQuery已经跟不上时代的步伐，因此一些新库如雨后春笋般出来，如fastclick、iscroll、react、fetch-polyfill、es5-shim、babel、rollup、rxjs …… 在react新版刚完成之际，react的版本号已经飙升到15，nodejs已经发展到8.0，这么多新东西，这么快的更新换代，一方面反映了前端技术的欣欣向荣，另一方面说明这个市场还不成熟。市场不成熟，正是框架高手“称雄一

方”的好时机。远的不说，就说国内的玉伯，由于适时推出seajs，与国外的requirejs竞争，国内舆论哗然，贬褒不一，但不管你说什么，时势造英雄，为什么英雄不是自己呢。说到底，实力很重要。

在前端最能反映威力的技术就是框架，其中一个评价标准是在GitHub上拿到很多星星的开源框架（网友的评论）。这是一本讲JavaScript框架的书，初版发布时，是国内唯一一本深入研究前端框架的书。新的一版，也是市面上唯一能把框架研究得较深入的书。深入并不代表难，但肯定有门槛。因为从2015年起，JavaScript就加速添加新特征、新语法、新功能，框架也变得越来越庞大，越来越复杂，因为对应的行业需求总是比我们的框架更复杂。是我们的框架适应现实，不是我们的框架无端变得如此“不可理喻”，学习门槛越来越高，要读懂已有框架的难度系数越来越陡峭。

当然，你也可以尝试无师自通，在武侠小说中，无师自通的主角不在少数，但无书自通者恐怕寥寥无几。我觉得有效的方法是多看几本进阶的书。有人说，书买十本看一本也值的，我觉得有待商榷。因此我推荐钱穆的另一个说法。当年，李敖向他请教治学方法，他回答说：并没有具体方法，要多读书、多求解，当以古书原文为底子、为主，免受他人成见的约束。书要看第一流的，一遍一遍读。与其十本书读一遍，不如一本书读十遍。不要怕大部头的书，养成读大部头书的习惯，则普通书就不怕了 。从这里面得到许多启示，一是读好书，二是读原版，三是精读，四是能读厚书。

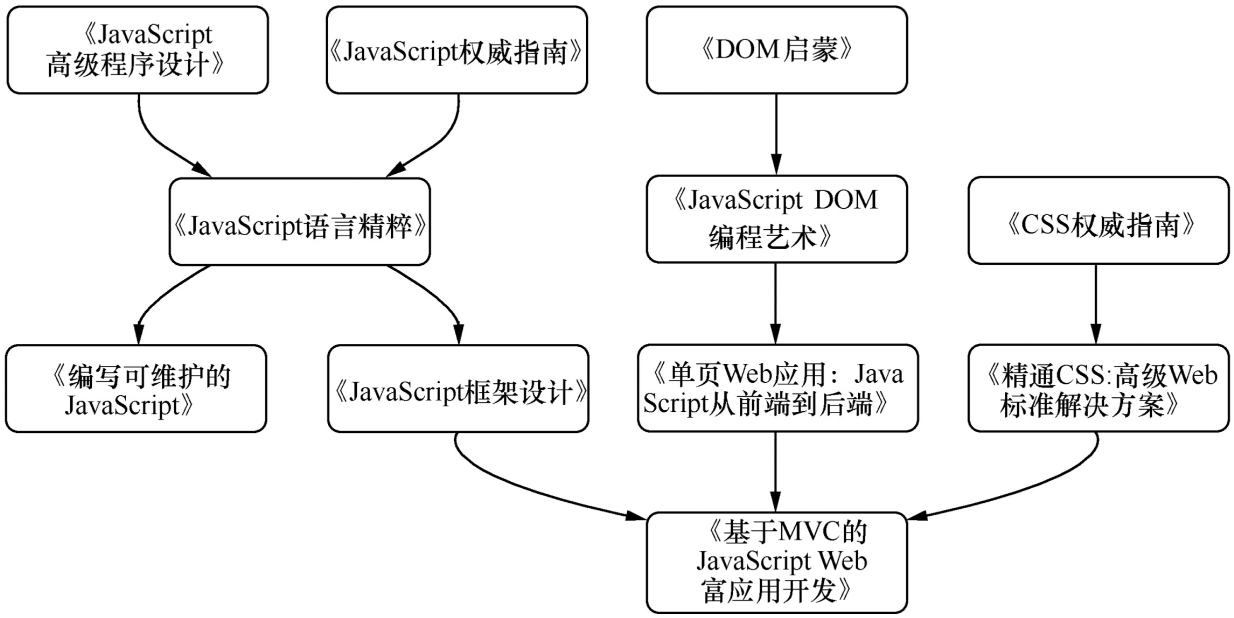
第1点读好书。在知乎，总是有新人请教学习前端该看什么书的问题。其实前端的好书也有不少，如红皮书、蝴蝶书、犀牛书、乌龟书、时钟书、鸽子书……《你不知道的JavaScript》，也很有

特色。

第2点读原版，在IT行业，特指英文原版。大凡有新技术出来，它们几乎都是以英文为载体出现，然后隔一年半载，才翻译到国内，所以学习新技术还是先看原版书。

第3点精读，就是重复看。多看书是一定的，但如果发现书上说的东西你始终看不进去时，一定不要坚持，一般来说看不下去是因为：一、你还没有达到看本书的水平，不适合你看；二、你看的书的确是本“烂书”，遇到这种情况赶紧换书。如果找到一本适合自己的书一定要坚持看下去，看到不懂的东西多搜索，搜索不到就多问人。当初，我阅读jQuery源码也是如此，最初看的是1.4.3版本，看得一头雾水，一气之下，从最初的1.0版本开始看。看完所有版本，了解其迭代过程，明白了一些代码后来为什么重构成这样。在那本书中，有相当多的源

码，里面着着实实比较了程序的演化过程。



第4点看厚书，一些集大成者的书，比如你创作文学，就看看《文心雕龙》；研究哲学，就得看黑格尔的《哲学全书》。在JavaScript领域，所谓的大部头莫过于《JavaScript权威指南》等。厚达千页，许多细节都涉及了。

看了这样的书，也不能保证你能写出漂亮的代码，更别说创造一个热门的框架。这情形就好像你学了些单词和语法，你就想听懂不带字幕的英文电影一样，那是不可能的。不同阶段，你需要掌握的技能是不一样的。框架是另一个层次的技术。但别

灰心，本书就是这些知识应用的指南，从高人的博客中、浏览器的更新日志中、GitHub的ISSUE中，辛辛苦苦整理来的知识，加上自己的实践分享给大家。

第一版的《JavaScript框架设计》是完全按照一个框架的编写顺序来编排，导致阅读难度太大，让许多读者反馈说无法坚持下去。这次大改版，将一些难点挪到后面，并加上大量的流程图与示意图来降低学习难度。并且随着浏览器的升级，许多东西也该更新换代，因此本书俨然是一本新书，无论是目录结构与内容上都焕然一新。最开始的知识是偏底层，讲授一些工具方法，一些常出现在各种框架的模块，里面会大量列举实现相同功能的不同实现，比如说isArray、trim、一个迷你的事件系统、获取元素样式、如何跨域……但越到后面，代码就越少，却能通过提供的网上链接，到相关博客里去观看相关的内容。后面几章都是非常前沿的东西，

许多技术方案还没有定案（至少在我成书时），没有最佳实践，只能提供一些抽象的方法论，让读者们自行悟道。有些东西不是死记硬背就能理解的。并且，本书也不是让你造一个avalon/angular的框架。每个时代对框架的需求都不一样。引用《从0到1》里的话——商业世界的每一刻都不会重演。下一个比尔·盖茨不会再开发操作系统，下一个拉里·佩奇或是谢尔盖·布林不会再研发搜索引擎，下一个马克·扎克伯格也不会去创建社交网络。如果你照搬这些人的做法，你就不是向他们学习了。

最后，感谢那些帮我审稿的朋友们（以下排名不分先后）：

王方磊、曹学进、张劲松、刘可、贺文榜、方正、索平、熊小涛、周静文、余永鹏、杨忠业、刘伟、曹学进、黄建新。

还有经常伴随我的可口可乐、香飘飘奶茶、旺

旺仙贝、大白兔奶糖、肉松饼、徐福记、牛轧糖、牛肉粒、鱿鱼丝、小黄鱼、曲奇饼、德芙巧克力……它们也不分先后！

本书答疑QQ群为（471118627），本书编辑和投稿邮箱为zhangtao@ptpress.com.cn。

InfoQ访谈

InfoQ：请您介绍一下自己目前的工作职责，负责的项目情况。据说您是写小说出身的，能否简单介绍一下自己的工作经历？为什么选择进入前端领域？为什么对前端框架抱有极大的热情？

笔者：现在我在“去哪儿网”任前端架构师一职，带领大家搞react开发。react，特别是react native是现时最好的移动框架。而公司的重点也转移到移动端了，因此我们小组负责将移动端相关设置全部搭建起来，各种“坑”踩一遍，努力为业务线提供最坚实的支撑。

每个人的经历都很曲折，尤其是前端人员，许多人都觉得前端比较容易学，或者赚钱比较多，才进入这行业。而我比较幸运，编代码也是我大学时

的一个乐趣。因此不会像其他人，会出现动力不足的情况。当然我的兴趣还是很多的，比如写小说、建筑、考古、学日语、学动画、看科幻小说、陶艺……会找乐子的人，不会轻易泄气。

总的来说，我的兴趣都有个特点，就是缺少与他人的互动。编代码也一样，一个人静静地编就好了。像PM，需要与别人不断沟通，我可能做不过来。每个人的性格都不一样，因此选择行业要符合自己的性格。

至于为什么进入前端，纯粹是偶然。我的一个弟弟是做这行的。我在小县城呆着赚不了几个钱，他说带我去深圳见识见识，便一下子介绍到他公司做前端了。我在小县城时也用前端接些小活过日子，因此不会觉得一下子跳跃太大。更重要的是，我特别能写程序，我没进入这行时，就写了三百多篇有关前端的博文，那时大家都以为我在大公司任

职了。

出于这样的误解，公司一开始就把一些很重要的事让我做，我要确保代码的质量，因为我的组件是被许多人复用的，就从那时起，我就一直搞框架、搞组件、搞各种工具。

infoQ：请您跟大家讲一下前端框架的发展历史、前端框架的起源和发展如何？现在的前端框架很多，其背后的原因是什么？国内的前端框架又是怎样起步的、发展现状如何？

笔者：这是一个老生常谈的话题，基本每本JavaScript书都会聊一下这个话题。主要原因是，JavaScript没有自己的SDK（核心库），需要依赖“民间”的力量。最开始是一些大公司有能力开发这些框架，如Prototype.js，也是作为ROR的次要项目开发出来的。此外还有dojo、closure、YUI这巨大的框架，也是大公司搞的。后来突然出现jQuery

这样由天才开发的框架。事实证明，大公司那一套管理方式，以KPI驱动的框架有着致命的缺憾，虽然面面俱到，但不能迅速吸收IT社区的新东西，使用起来不够方便灵活。

再后来，大家都知道是jQuery的天下，大家都争先恐后地为它做插件。jQuery也大大解放了程序员的生产力，让我们有时间做一些更有意义的事。在后jQuery时代，最有意义的两件事是RequireJS的诞生与nodejs的出世。前者试图解决JavaScript模块化问题，后者让我们能从后端那里“抢”走一些活儿，那些活儿本来就是前端做比较合适。比如说做模板、套模板、传数据、JavaScript的语法检测、风格检测、理点等。

这个时间里，产生了像backbone这样的MVC框架。但立即被knockout、angular、react等MVVM框架占去份额了。要知道，后端从MVC进化到

MVVM，需要大概10年时间，而前端则不到2年。前端框架发展实在太迅猛了！

我想其背后最大的动力是需求！源源不断的需求！原来由后端做的活儿，放在前端做更合适、更快，用户体验更好。这是趋势使然，挡也挡不住。

目前国内的发展历程其实与国外一模一样。最开始是公司牵头，后来就涌现大量出色的个人项目。阿里的前端技术之所以这么强，是因为他们不断地研制自己的“轮子”，“轮子”会越造越好。那些绝不重复造“轮子”的人默默无名，而框架作者们则开创自己一片新天地了。中国拥有世界上最庞大的互联网市场，面临的挑战很大，光是满足国内用户的需求，国内框架的研发人员就需要比国外同行更加努力。经过这几年的磨练，国内的框架也渐渐走出国门（如我的avalon，在澳大利亚、德国都有人在用，又如百度的ECharts，这个也非常抢眼）。

infoQ：**avalon**的起源与发展是如何的？**avalon 2**的架构如何？采用这样的架构有什么好处？与其他框架相比，**avalon**更加**“**接地气**”**的点体现在哪些地方？

笔者：avalon当初只是我另一个早期的框架mass Framework的一个插件。mass Framework类似于jQuery与Prototypejs的结合体。没什么特色，被埋没也是必然的。但我说过，“轮子”会越造越好的。当我将这个插件介绍到博客园 ——国内一个非常著名的IT社区，反映很不错。于是独立出来搞。经过5年的发展，它渐渐拥有自己的论坛与社区。不过，由于年龄的增长，我也开始抗拒一些新东西（比如说社区上一些自动化工具，总是想自己全部实现），导致avalon一度发展缓慢。发展到1.5版时试图奋起直追，效果不明显。avalon2决定使用一个更吸引眼球的东西扭转局面——这就是虚拟DOM，带来了性能上的飞跃。MVVM虽然非常方

便，但很容易出现性能瓶颈。出自于谷歌之手的angular，也有2000个指令（即一个页面超过2000个指令，页面更新就慢得令人发指）。Facebook的react带来了“虚拟DOM”这个新概念，使用轻量对象代替重型对象来承担绝大多数的页面重绘工作，解决了所谓的“性能墙”问题。

原来MVVM架构分3层，M、V、VM三层，我们只需要关注VM。VM通过各种手段得知外界对它的操作，然后它智能地通知M与V进行变更。VM承受太多职责，导致不堪重负。而虚拟DOM的导入，让avalon2拥有4层架构。虚拟DOM位于V与VM之间，复杂的视图计算由虚拟DOM计算好，然后diff出差异点实现最小化刷新。这是算法的伟大胜利。为了实现虚拟DOM，前端框架作者也接触编译原理等高深的东西了。

现在主流的MVVM也结合虚拟DOM进行性能

优化。基本上它们是基于Object.defineProperty这个API。而这个API在IE8中有bug，只能用于IE9＋。因此它们的兼容性都比较差。而avalon的优势在于其作者精通各种兼容性问题与“魔法”。在IE6～IE8下，我找到了VBScript实现对VM的自省机制，在较新的浏览器使用Object.defineProperty，在更新锐的浏览器，则使用Proxy（动态代理）这个划时代的东西，从此我们可以动态监听对象是否被添加删除了某个属性，或调用了某个方法，而不像Object.defineProperty只能监听读写操作（Proxy对象被用于定义自定义基本操作的行为，如属性查找、分配、枚举、函数调用等，在Firefox的定义中一共有14个属性）。

从上面的描述来看，avalon走在时代的前列，但它不忘初心，还继续支持IE6，让大家用MVVM或虚拟DOM时没有后顾之忧。并且大家也不用担心avalon为了兼容IE6，会变得非常冗长。因为

avalon是一份源码编译出好几个版本，每个版本根据浏览器的支持程度合并对应的模板。如果只想运行于IE10，其会相当小。

infoQ：在选择前端框架时，大家的建议很多，例如结合自己的业务等。您也曾提到，选择前端框架应综合考虑框架本身与团队情况。要考虑的点这么多，究竟怎样来综合考虑呢？具体的步骤应该是怎样的？

笔者：的确如此，技术本来是为业务服务的，单纯“玩”技术是没有前途，也找不到方向。前端框架之所以这么多，也是因为大家的业务侧重点各不相同。选择合适的框架，比选择一个时髦的框架重要多了。千万别让手下人员自行决定。他们玩不转可以“拍拍屁股走人”，留下一个烂摊子。我们要考虑到业务的可持续性、代码的可交接性及团队的普遍接受能力。比如一个公司，没有前端开发人员，

都是后端开发人员顺手做前端的活，早期许多公司都是招PHP实现前端开发。他们的设计模式比较好，可以上手angular。如果一个团队新人够多，不稳定，则只能用jQuery与bootstrap。如果是一个创业公司，急着做出原型来拉投资，可以尝试vue、avalon、react等短平快的框架。但我所说的还是核心框架，涉及图表、UI库，这些需要架构师见多识广，自己“趟过坑”，才让团队“集中过河”。

infoQ：有人说前端编程标准和方法渐渐出现稳定的趋势，您怎么看待这一观点？在之后的发展过程中，有没有可能标准完全统一？有没有可能某个前端框架**“**一统江湖**”**？

笔者：这个观点前半段是对的。像jQuery带来一系列便捷的操作DOM的方式，append、prepend、remove等方法已经在DOM4中实现了。其最著名的选择器引擎，也有了原生替代品。因为浏

览器商之间也存在竞争关系，将一些公认的好东西内置可以讨好用户。但浏览器厂商之间没怎么沟通，W3C给出的规范也模糊，出现差异是在所难免的。因此不要相信浏览器，要使用框架！至于框架，框架之争是不会停息的，好的框架会不断涌现，它们可能以某个神奇的设计一下推翻前面的技术。就像jQuery“灭掉”prototype，gulp“灭掉”grunt，webpack“灭掉”browserfy，react“灭掉”angular ……

并且没有一个框架覆盖所有需求，每个领域都有领头羊。因此想一个框架“一统江湖”，这个不可能也不实际。只要做好自己的擅长之处，开发就会比较顺利。

infoQ：您认为，前端开发人员学习框架设计应具备哪些能力？应从哪些方面着手进行设计？哪些地方有“坑”，需要注意避开？

笔者：这个问题比较笼统，我也只能笼统地回

答。就像你问怎么挣大钱，有许多东西，人家说出来你也不能复制。首先，基础很重要，如计算机科班出来的人，搞前端就很容易上手。其次是设计模式，这是Java十多年积累的精华，是我们构建巨型工程的利器。现在前端框架的程序很多是上万行了。像过去那样，全是方法＋全局变量在堆砌，在生产环境中找bug是恶梦。最后是好好看高手们的框架，阅读源码是进步最快的方式之一。只有看了足够多的源码，你才能博采众长。再最后，就是宣传与测试了，宣传确保你拥有第一批用户，成为你继续维护与升级的动力。需要提供一系列便捷的下载渠道，因为酒香也怕巷子深。测试是确保你能留住用户。目前社区上有大量的测试工具，你可以将它们全部绑定在webpack，在用户build工程时，把所有测试运行一遍。

最后说“坑”，其实没什么“坑”，所有浏览器兼容性问题与技术难点，许多技术高人都提供现成方

案。但如果你做出框架，不再维护，对使用者来说这才是“大坑”。就是你不想维护了，就要找一个接盘者来主持。就像jQuery、nodejs等著名项目，原作者早早交给他人维护。

我今天的分享就到这里，谢谢大家！

第1章　种子模块



1.1 模块化

最近几年，JavaScript得到飞速发展，一些框架越来越大，已经不像过去那样全部写进一个JS文件中。但拆到多个JS文件时，就要决定哪个是入口文件，哪个是次要文件，而这些次要文件也不可能按1、2、3、4的顺序组织起来，可能1依赖于2、3，3依赖于4、5，每个文件的顶部都像其他语言那样声明其依赖，最后在结束时说明如何暴露出那些变量或方法给外部使用。

就算你的框架只有几千行，在开发时将它们按功能拆分为10多个文件，维护起来也非常方便。加之Node.js的盛行，ES2016许多语法不断被浏览器支持，我们更应该拥抱模块化。

本书所介绍的所有模块，都以Node.js提倡的CommonJS方式组织起来。

时下流行3种定义模块的规范：AMD [1] 、CommonJS [2] 与ES6 module。它们都被webpack所支持。以AMD定义JS模块通过RequireJS能直接运行于浏览器；CommonJS则需要browserfy等Node.js打包后才能运行于浏览器；ES6 module在我写书时，还没有浏览器支持，需要webpack、rollup等Node.js工具打包才能运行于浏览器。

下面简略演示一下这3种模块的定义方式。

```
//AMD
define(['./aaa', './bbb'], function(a, b){
   return {
       c: a + b
   }
})

// CommonJS
var a = require('./aaa')
var b = require('./bbb')
module.exports = {
    c: a + b
}

//es6 module
import a form './aaa'
import b form './bbb'
var c = a + b
export {c}
```

有关这3种模块的用法，后面的章节会详述，这里暂时略过。

本人是比较倾向使用CommonJS的，因为其相关的打包工具非常成熟，并且以这种方式编写的框架，可以与大量Node.js编写的测试框架无缝配合使用。

再说回来，刚才提到的入口文件。整个程序就是引入其他子模块，然后导出代表命名空间的JavaScript对象就行了。

```
var avalon = require('./seed/index')

require('./filters/index')
require('./vdom/index')
require('./dom/index')
require('./directives/index')
require('./strategy/index')
require('./component/index')
require('./vmodel/index')

module.exports = avalon
```

我们编写一个框架时，可能拆成上百个JS文

件。为了方便寻找，这时需要按照功能或层次放进不同的子文件夹。然后每个子文件都有它的入口文件（index.js），由它来组织其依赖。本章的种子模块，就放到seed这个文件夹下了。

1.2　功能介绍

种子模块 也叫核心模块 ， 是框架的最先执行的部分。即便像jQuery那样的单文件函数库，它的内部也分许多模块，必然有一些模块冲在前面立即执行；有一些模块只有用到才执行；也有一些模块（补丁模块 ）可有可无，存在感比较弱，只在特定浏览器下才运行。

既然是最先执行的模块，那么就要求其里面的方法是历经考验、千锤百炼的，并且能将这个模块变得极具扩展性、高可用、稳定性。

（1）扩展性 ， 是指方便将其他模块的方法或属性加入进来，让种子迅速成长为“一棵大树”。

（2）高可用 ， 是指这里的方法是极其常用的，其他模块不用重复定义它们。

（3）稳定性，是指不能轻易在以后版本中删除，要信守承诺。

参照许多框架与库的实现，笔者认为种子模块应该包含如下功能：对象扩展、数组化、类型判定、无冲突处理、domReady。本章讲解的内容以avalon种子模块为范本，可以在下面的地址中阅览。

```
https://github.com/RubyLouvre/avalon/tree/master/src/seed
```

1.3　对象扩展

我们需要一种机制，将新功能添加到我们的命名空间上。命名空间，是指我们这个框架在全局作用域暴露的唯一变量，它多是一个对象或一个函数。命名空间通常也就是框架名字。我们可以看一下别人是如何为框架起名字的。

```
https://www.zhihu.com/question/46804815
```

回到主题，对象扩展这种机制，我们一般做成一个方法，叫做extend或mixin。JavaScript对象在属性描述符[3]（Property Descriptor）没有诞生之前，是可以随意添加、更改、删除其成员的，因此扩展一个对象非常便捷。一个简单的扩展方法实现如下。

```
//Prototype.js
function extend(destination, source) {
    for (var property in source)
```

```
        destination[property] = source[property];
    return destination;
}
```

不过，旧版本IE在这里有个问题，它认为像Object的原型方法就是不应该被遍历出来，因此for in循环是无法遍历名为valueOf、toString的属性名。这导致，后来人们模拟Object.keys方法实现时也遇到了这个问题。

```
Object.keys = Object.keys || function(obj){
    var a = [];
    for(a[a.length] in obj);
    return a ;
}
```

在不同的框架中，这个方法还有不同的实现，如EXT分为apply与applyIf两个方法，前者会覆盖目标对象的同名属性，而后者不会。dojo允许多个对象合并在一起。jQuery还支持深拷贝。下面是avalon.mix方法，几乎是照搬jQuery.extend方法，只是在VBscript上做了一些防御处理。

```
avalon.mix = avalon.fn.mix = function () {
    var options, name, src, copy, copyIsArray, clone,
            target = arguments[0] || {},
            i = 1,
            length = arguments.length,
            deep = false

    // 如果第一个参数为布尔,判定是否深拷贝
    if (typeof target === 'boolean') {
        deep = target
        target = arguments[1] || {}
        i++
    }

    //确保接受方为一个复杂的数据类型
    if (typeof target !== 'object' && !avalon.isFunction(target
)) {
        target = {}
    }

    //如果只有一个参数，那么新成员添加于mix所在的对象上
    if (i === length) {
        target = this
        i--
    }

    for (; i < length; i++) {
        //只处理非空参数
        if ((options = arguments[i]) != null) {
            for (name in options) {
                try {
                    src = target[name]
                    copy = options[name] //当options为VBS对象时报
错
                } catch (e) {
                    continue
                }

                // 防止环引用
                if (target === copy) {
                    continue
                }
                if (deep && copy && (avalon.isPlainObject(copy)
 || (copyIsArray = Array.
                isArray(copy)))) {
```

```
                    if (copyIsArray) {
                        copyIsArray = false
                        clone = src && Array.isArray(src) ? src
 : []

                    } else {
                        clone = src && avalon.isPlainObject(src
) ?

                        src : {}
                    }

                    target[name] = avalon.mix(deep, clone, copy
)
                } else if (copy !== void 0) {
                    target[name] = copy
                }
            }
        }
    }
    return target
}
```

代码里面用到了avalon.isFunction, avalon.isPlainObject，它们也属于种子模块的一部分，下面有解说。

由于此功能这么常用，到后来ES6就干脆支持它了，于是有了 **Object.assgin** 。如果要低端浏览器直接用它，可以使用以下polyfill [4] 。

```
function ToObject(val) {
```

```
    if (val == null) {
        throw new TypeError('Object.assign cannot be called wit
h null or undefined');
    }

    return Object(val);
}
module.exports = Object.assign || function (target, source) {
    var from;
    var keys;
    var to = ToObject(target);

    for (var s = 1; s < arguments.length; s++) {
        from = arguments[s];
        keys = Object.keys(Object(from));

        for (var i = 0; i < keys.length; i++) {
            to[keys[i]] = from[keys[i]];
        }
    }

    return to;
};
```

1.4　数组化

浏览器下存在许多类数组对象，如function内的arguments，通过document.forms、form.elements、doucment.links、select.options、document.getElementsByName、document.getElementsBy TagName、childNodes、children等方式获取的节点集合（HTMLCollection、NodeList），或依照某些特殊写法的自定义对象。

```
var arrayLike = {
    0: "a",
    1: "1",
    2: "2",
    length: 3
}
```

类数组对象 是一个很好的存储结构，不过功能太弱了，为了享受纯数组的那些便捷方法，我们在处理它们前都会做一下转换。

通常来说，使用Array.prototype.slice.call就能转换我们的类数组对象了，但旧版本IE下的HTMLCollection、NodeList不是Object的子类，采用如上方法将导致IE执行异常。我们看一下各大库怎么处理的。

```
//jQuery的makeArray
var makeArray = function(array) {
    var ret = [];
    if (array != null) {
        var i = array.length;
        // The window, strings (and functions) also have 'lengt
h'
        if (i == null || typeof array === "string" || jQuery.is
Function(array) ||
        array.setInterval)
            ret[0] = array;
        else
            while (i)
                ret[--i] = array[i];
    }
    return ret;
}
```

jQuery对象是用来储存与处理dom元素的，它主要依赖于**setArray** 方法来设置和维护长度与索引，而setArray的参数要求是一个数组，因此makeArray的地位非常重要。这方法保证就算没有

参数也要返回一个空数组。

Prototype.js的$A方法如下：

```
function $A(iterable) {
    if (!iterable)
        return [];
    if (iterable.toArray)
        return iterable.toArray();
    var length = iterable.length || 0, results = new Array(leng
th);
    while (length--)
        results[length] = iterable[length];
    return results;
};
```

mootools的$A方法如下：

```
function $A(iterable) {
    if (iterable.item) {
        var l = iterable.length, array = new Array(l);
        while (l--)
            array[l] = iterable[l];
        return array;
    }
    return Array.prototype.slice.call(iterable);
};
```

Ext的toArray方法如下：

```
var toArray = function() {
    return isIE ?
            function(a, i, j, res) {
                res = [];
                Ext.each(a, function(v) {
                    res.push(v);
                });
                return res.slice(i || 0, j || res.length);
            } :
            function(a, i, j) {
                return Array.prototype.slice.call(a, i || 0, j
|| a.length);
            }
}()
```

Ext的设计比较巧妙，功能也比较强大。它一开始就自动执行自身，以后就不用判定浏览器了。它还有两个可选参数，对生成的纯数组进行操作。

但纵观这些方法，其实有一个特点，就是优化考虑使用Array.prototype.slice来转换类数组对象。而Array.prototype.slice碰到的唯一障碍就是旧版本IE下的节点集合。那么，我们设法让IE下的Array.prototype.slice能切割节点集合就一帆风顺了。以下是avalon的解决方案，不过大部分代码是来自firefox官方的。

```
//https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Refer
ence/Global_Objects/Array/slice
/**
* Shim for "fixing" IE's lack of support (IE < 9) for applying
slice
* on host objects like NamedNodeMap, NodeList, and HTMLCollecti
on
* (technically, since host objects have been implementation-dep
endent,
* at least before ES6, IE hasn't needed to work this way).
* Also works on strings, fixes IE < 9 to allow an explicit unde
fined
* for the 2nd argument (as in Firefox), and prevents errors whe
n
* called on other DOM objects.
*/

var _slice = Array.prototype.slice
try {
    // Can't be used with DOM elements in IE < 9
    _slice.call(document.documentElement)
} catch (e) { // Fails in IE < 9
    // This will work for genuine arrays, array-like objects,
    // NamedNodeMap (attributes, entities, notations),
    // NodeList (e.g., getElementsByTagName), HTMLCollection (e
.g., childNodes),
    // and will not fail on other DOM objects (as do DOM elemen
ts in IE < 9)
    Array.prototype.slice = function (begin, end) {
        // IE < 9 gets unhappy with an undefined end argument
        end = (typeof end !== 'undefined') ? end : this.length

        // For native Array objects, we use the native slice fu
nction
        if (Array.isArray(this) ) {
            return _slice.call(this, begin, end)
        }

        // For array like object we handle it ourselves.
        var i, cloned = [],
              size, len = this.length

        // Handle negative value for "begin"
        var start = begin || 0
        start = (start >= 0) ? start : len + start
```

```
        // Handle negative value for "end"
        var upTo = (end) ? end : len
        if (end < 0) {
           upTo = len + end
        }

        // Actual expected size of the slice
        size = upTo - start

        if (size > 0) {
            cloned = new Array(size)
            if (this.charAt) {
                for (i = 0; i < size; i++) {
                    cloned[i] = this.charAt(start + i)
                }
            } else {
                for (i = 0; i < size; i++) {
                    cloned[i] = this[start + i]
                }
            }
        }

        return cloned
    }
}

avalon.slice = function (nodes, start, end) {
    return _slice.call(nodes, start, end)
}
```

上面的Array.prototype.slice polyfill可以放到另一个补丁模块，这样确保我们的框架在升级时非常轻松地抛弃这些历史包袱。

1.5　类型的判定

JavaScript存在两套类型系统：一套是基本数据类型，另一套是对象类型系统。基本数据类型在ES5中包括6种，分别是undefined、string、null、boolean、function和object。基本数据类型是通过typeof来检测的。对象类型系统是以基础类型系统为基础的，通过instanceof来检测。然而，JavaScript自带的这两套识别机制非常不靠谱，于是催生了**isXXX**系列。就拿typeof来说，它只能粗略识别出string、number、boolean、function、undefined和object这6种数据类型，无法识别Null、RegExp和Argument等细分对象类型。

让我们看一下这里面究竟有多少陷阱。

```
typeof null// "object"
typeof document.childNodes //safari "function"
typeof document.createElement('embed')//ff3-10 "function"
typeof document.createElement('object')//ff3-10 "function"
typeof document.createElement('applet')//ff3-10 "function"
```

```
typeof /\d/i //在实现了ecma262v4的浏览器返回 "function"
typeof window.alert //IE678 "object""
var iframe = document.createElement('iframe');
document.body.appendChild(iframe);
xArray = window.frames[window.frames.length - 1].Array;
var arr = new xArray(1, 2, 3); // [1,2,3]
arr instanceof Array; // false
arr.constructor === Array; // false

window.onload = function() {
    alert(window.constructor);// IE67 undefined
    alert(document.constructor);// IE67 undefined
    alert(document.body.constructor);// IE67 undefined
    alert((new ActiveXObject('Microsoft.XMLHTTP')).constructor)
;// IE6789 undefined
}
isNaN("aaa") //true
```

上面分4组，第一组是typeof的坑。第二组是instanceof的陷阱，只要原型上存在此对象的构造器它就返回true，但如果跨文档比较，iframe里面的数组实例就不是父窗口的Array的实例。第三组是有关constructor的陷阱，在旧版本IE下，DOM与BOM对象的constructor属性是没有暴露出来的。最后有关NaN，NaN对象与null、undefined一样，在序列化时是原样输出的，但isNaN这方法非常不靠谱，把字符串、对象放进去也返回true，这对我们序列

化非常不利。

另外，在IE下typeof还会返回**unknown** 的情况。

```
if (typeof window.ActiveXObject != "undefined") {
    var xhr = new ActiveXObject("Msxml2.XMLHTTP");
    alert(typeof xhr.abort);
}
```

基于这IE的特性，我们可以用它来判定某个VBscript方法是否存在。

```
<script type="text/VBScript">
 function VBMethod(a,b)
 VBMethod = a + b
 end function
</script>

<script>
if(typeof VBMethod === "unknown"){//看这个
    alert(VBMethod(10,34))
}
</script>
```

另外，以前人们总是以document.all是否存在来判定IE，这其实是很危险的。因为用document.all来

取得页面中的所有元素是不错的主意，这个方法Firefox、Chrome觊觎好久了，不过人们都这样判定，于是有了在Chrome下的这出“闹剧”。

```
typeof document.all // undefined
document.all // HTMLAllCollection[728] (728为元素总数)
```

在判定undefined、null、string、number、boolean和function这6个还算简单，前面2个可以分别与void（0）、null比较，后面4个直接typeof也可满足90%的情形。这样说是因为string、number、boolean可以包装成“伪对象”，typeof无法按照我们的意愿工作了，虽然它严格执行了Ecmascript的标准。

```
typeof new Boolean(1);//"object"
typeof new Number(1);//"object"
typeof new String("aa");//"object"
```

这些还是最简单的，难点在于RegExp与

Array。判定RegExp类型的情形很少，不多讲了，Array则不一样。有关isArray的实现方法不下20种，都是因为JavaScript的鸭子类型 被攻破了。

以下代码是isArray早些年的探索。

```
function isArray(arr) {
    return arr instanceof Array;
}

function isArray(arr) {
    return !!arr && arr.constructor == Array;
}

function isArray(arr) {//Prototype.js1.6.0.3
    return arr != null && typeof arr === "object" &&
            'splice' in arr && 'join' in arr;
}

function isArray(arr) {//Douglas Crockford
    return typeof arr.sort == 'function'
}

function isArray(array) {//kriszyp
    var result = false;
    try {
        new array.constructor(Math.pow(2, 32))
    } catch (e) {
        result = /Array/.test(e.message)
    }
    return result;
};

function isArray(o) {// kangax
    try {
        Array.prototype.toString.call(o);
        return true;
    } catch (e) {
```

```
        }
    return false;
};

function isArray(o) {//kangax
    if (o && typeof o == 'object' && typeof o.length == 'number
' && isFinite(o.length)) {
        var _origLength = o.length;
        o[o.length] = '__test__';
        var _newLength = o.length;
        o.length = _origLength;
        return _newLength == _origLength + 1;
    }
    return false;
}
```

至于null、undefined、NaN则比较好对付。

```
function isNaN(obj) {
    return obj !== obj
}
function isNull(obj) {
    return obj === null;
}
function isUndefined(obj) {
    return obj === void 0;
}
```

这一切烦恼，直到Prototype.js把Object.prototype.toString发掘出来才被彻底克服。此方法是直接输出对象内部的[[Class]]，绝对精准。

有了它，可以跳过95%的陷阱了。当然，**toString** 方法 只能针对原生数据类型，像如何检测是否为window、是否为纯净JavaScript对象，就有点力不从心。

由于ECMA是不规范Host对象，window对象属于Host，所以也没有被约定，就算Object.prototype.toString也对它无可奈何。

```
[object Object] IE6
[object Object] IE7
[object Object] IE8
[object Window] IE9
[object Window] firefox3.6
[object Window] opera10
[object DOMWindow] safai4.04
[object global] chrome5.0.3.22
```

不过根据window.window和window.setInterval去判定更加不靠谱，用一个技巧我们可以完美识别IE6、IE7、IE8的window对象，其他还是用toString。这个神奇的hack（技巧）就可以，window与document互相比较，如果顺序不一样，其

结果是不一样的！

```
window == document // IE678 true;
document == window // IE678 false;
```

当然，如果细数起来，JavaScript匪夷所思的事比比皆是。

```
存在a !== a的情况；
存在a == b && b != a的情况；
存在a == !a的情况；
存在a === a+100的情况；
1 < 2 < 3为true，3 > 2 > 1为false；
0/0为NaN；
……
```

另一个比较常用的判定isArrayLike方法，这用于遍历方法，通常叫做each，它可能循环对象与类数组对象。

好了，我们看一下各大框架有多少is×××方法，哪些是最常用的，方便我们的框架也能包含它们，如表1-1所示。

表**1-1**

框架	方法
Prototype.js	isElement、isArray、isHash、isFunction、isString、isNumber、isDate、isUndefined
mootools	typeOf判定基本类型，instanceOf判定自定义“类”
RightJS	isFunction、isHash、isString、isNumber、isArray、isElement、isNode
EXT	isEmpty、isArray、isDate、isObject、isSimpleObject、isPrimitive、isFunction、isNumber、isNumeric、isString、isBoolean、isElement、isTextNode、isDefined、isIterable
Underscore.js	isElement、isEmpty、isArray、isArguments、isObject、isFunction、isString、isNumber、isFinite、isNaN、isBoolean、isDate、isRegExp、isNull、isUndefined
jQuery	isFunction、isArray、isPlainObject、isEmptyObject、isWindow、isNumeric、isNaN、isXML
avalon	isFunction、isObject、isPlainObject、isWindow、isVML

1.5.1 type

我们比较一下，发现jQuery针对基础类型的

isXXX是很少的，只是isFunction与isArray。其他都是很特别的业务判定。究其原因，是因为jQuery发明type方法，这个方法就囊括了isBoolean、isNumber、isString、isFunction、isArray、isDate、isRegExp、isObject及isError。

```
//jquery2.0
var class2type
// Populate the class2type map
jQuery.each("Boolean Number String Function Array Date RegExp O
bject Error".split(" "), function(i, name) {
    class2type[ "[object " + name + "]" ] = name.toLowerCase();
});

jQuery.type = function( obj ) {
    if ( obj == null ) {
        return String( obj );
    }
   // Support: Safari <= 5.1 (functionish RegExp)
    return typeof obj === "object" || typeof obj === "function"
 ?
            class2type[ core_toString.call(obj) ] || "object" :
            typeof obj;
}
```

这个方法也不断进化，因为ES6也加入了新的数据类型Symbol。

此外，isXXX系列随着框架版本的提升，也会

就像恶性肿瘤一样不断膨胀，其实你多弄几个isXXX也不能满足用户的全部需求。就像isDate、isRegExp会用到的概率有多高呢？

下面我们回溯一下jQuery的发展历程，看它是怎么扩张自己的isXXX系列的。

1.5.2 isPlainObject

在jQuery 1.4中只有isFunction、isArray、isPlainObject和isEmptyObject。IsFunction和isArray肯定是用户用得最多。isEmptyObject是用于数据缓存系统，当此对象为空时，就可以删除它。isPlainObject则是用来判定是否为纯净的JavaScript对象，既不是DOM、BOM对象，也不是自定义“类”的实例对象，制造它的最初目的是用于深拷贝，避开像window那样自己引用自己的对象。

```
//jQuery2.0
jQuery.isPlainObject = function(obj) {
    //首先排除基础类型不为Object的类型，然后是DOM节点与window对象
    if (jQuery.type(obj) !== "object" || obj.nodeType || jQuery
```

```
.isWindow(obj)) {
        return false;
    }
    //然后回溯它的最近的原型对象是否有isPrototypeOf,
    //旧版本IE的一些原生对象没有暴露constructor、prototype，因此会在这
里过滤
    try {
        if (obj.constructor &&
                !hasOwn.call(obj.constructor.prototype, "isProt
otypeOf")) {
            return false;
        }
    } catch (e) {
        return false;
    }
    return true;
}

//jQuery2.2 变得更加严密，并且放弃try catch以提高性能
jQuery.isPlainObject = function(obj) {
        var key;
        if ( jQuery.type( obj ) !== "object" || obj.nodeType ||
 jQuery.isWindow( obj ) ) {
            return false;
        }

        // Not own constructor property must be Object
        if ( obj.constructor &&
                !hasOwn.call( obj, "constructor" ) &&
                !hasOwn.call( obj.constructor.prototype || {},
"isPrototypeOf" ) ) {
           return false;
        }

        // Own properties are enumerated firstly, so to speed u
p,
        // if last one is own, then all properties are own
        for ( key in obj ) {}

        return key === undefined || hasOwn.call( obj, key );
}

//jQuery3.0 放弃对IE6-8的支持，因此可以直接用Object.getPrototypeOf方
法
            //并且也无需特殊处理window
```

```
var fnToString.call = hasOwn.toString;
var ObjectFunctionString  = fnToString.call( Object );
var toString = class2type.toString;
var hasOwn = class2type.hasOwnProperty;
var getProto = Object.getPrototypeOf;
jQuery.isPlainObject = function( obj ) {
       var proto, Ctor;

       // Detect obvious negatives
       // Use toString instead of jQuery.type to catch host obj
ects
       if ( !obj || toString.call( obj ) !== "[object Object]"
) {
           return false;
       }

       proto = getProto( obj );

       // Objects with no prototype (e.g., `Object.create( null
 )`) are plain
       if ( !proto ) {
           return true;
       }

       // Objects with prototype are plain iff they were constr
 ucted by a global Object function
       Ctor = hasOwn.call( proto, "constructor" ) && proto.cons
tructor;
       return typeof Ctor === "function" && fnToString.call( Ct
or ) === ObjectFunctionString;
```

在avalon中有一个更精简的版本，由于它只支持IE10等非常新的浏览器及不支持跨iframe，就没有干扰因素了，可以大胆使用ecma262v5的新API。

```
avalon.isPlainObject = function(obj) {
    return typeof obj === "object" && Object.getPrototypeOf(obj
```

```
) === Object.prototype
}
```

补充一句，1.3版本中，Prototype.js的研究成果（Object.prototype.toString.call）就应用于jQuery了。在1.2版本中，jQuery判定一个变量是否为函数非常复杂。

```
jQuery.isFunction = function( fn ) {
    return !!fn && typeoffn != "string" && !fn.nodeName &&
    fn.constructor != Array && /^[\s[]?function/.test( fn + ""
);
 }
```

1.5.3　isWindow

jQuery 1.43引入isWindow来处理makeArray中对window的判定，引入isNaN用于确保样式赋值的安全。同时引入type代替typeof关键字，用于获取数据的基本类型。

```
var isWindow = function( obj ) {
    return obj != null && obj === obj.window;
}
```

```
```

jQuery这个判定是非常粗糙，通过不了伪造的对象。

```
var fakeWindow = {}
fakeWindow.window = fakeWindow
$.isWindow(fakeWindow) //true，这里骗过jQuery了！！！
```

对比一下avalon的实现 [5] 。

```
avalon.isWindow = function (obj) {
    if (!obj)
        return false
    // 利用IE6、IE7、IE8 window == document为true,document == window竟然为false的神奇特性
    // 标准浏览器及IE9、IE10等使用正则检测
    return obj == obj.document && obj.document != obj
}

var rwindow = /^\[object (?:Window|DOMWindow|global)\]$/
function isWindow(obj) {//现代浏览器使用这个实现
    return rwindow.test(toString.call(obj))
}

if (isWindow(window)) {
    avalon.isWindow = isWindow
}
```

1.5.4　isNumeric

jQuery1.7中添加isNumeric代替isNaN。这是个不同于其他框架的isNumber，它可以是字符串，只要外观上像数字就行了。但jQuery1.7还做了一件违背之前提到稳定性的事情，贸然去掉jQuery.isNaN，因此导致基于旧版本jQuery的一大批插件失效。

```
//jquery1.43~1.64
jQuery.isNaN = function(obj) {
    return obj == null || !rdigit.test(obj) || isNaN(obj);
})
//jquery1.7 就是isNaN的取反版
jQuery.isNumeric = function(obj) {
    return obj != null && rdigit.test(obj) && !isNaN(obj);
})
//jquery1.71~1.72
jQuery.isNumeric = function(obj) {
    return !isNaN(parseFloat(obj)) && isFinite(obj);
}
//jquery2.1
jQuery.isNumeric = function(obj) {
    return obj - parseFloat(obj) >= 0;
}
```

1.5.5 isArrayLike

此外，jQuery还有一个没有公开的isXXX——

isArrayLike。由于一个类数组太难了，唯一的辨识手段是它应该有一个大于或等于零的整型length属性。此外，还有一些“共识”，如window与函数和元素节点（如form元素）不算类数组，虽然它们都满足前面的条件。因此，至今jQuery也没有把它暴露出来。

```
//jquery2.0
function isArraylike(obj) {
    var length = obj.length, type = jQuery.type(obj);
    if (jQuery.isWindow(obj)) {
        return false;
    }
    if (obj.nodeType === 1 && length) {
        return true;
    }
    return type === "array" || type !== "function" &&
           (length === 0 ||
                   typeof length === "number" && length > 0 &&
(length - 1) in obj);
}

//jQuery3.0
function isArrayLike( obj ) {

    // Support: real iOS 8.2 only (not reproducible in simulato
r)
    // 'in' check used to prevent JIT error (gh-2145)
    // hasOwn isn't used here due to false negatives
    // regarding Nodelist length in IE
    var length = !!obj && "length" in obj && obj.length,
        type = jQuery.type( obj );

    if ( type === "function" || jQuery.isWindow( obj ) ) {
        return false;
```

```
    }

    return type === "array" || length === 0 ||
        typeof length === "number" && length > 0 && ( length -
1 ) in obj;
}
```

avalon也独立发展出自己的isArrayLike方法，但也不敢暴露出来。

```
//avalon 1.4
var toString = class2type.toString
var rarraylike = /(Array|List|Collection|Map|Arguments)\]$/
var rfunction = /^\s*\bfunction\b/
function isArrayLike(obj) {
    if (!obj)
        return false
    var n = obj.length
    if (n === (n >>> 0)) { //检测length属性是否为非负整数
        var type = toString.call(obj).slice(8, -1)
        if (rarraylike.test(type))
            return false
        if (type === 'Array')
            return true
        try {
            if ({}.propertyIsEnumerable.call(obj, 'length') ===
 false) { //如果是原生对象
                return rfunction.test(obj.item || obj.callee)
            }
            return true
        } catch (e) { //IE的NodeList直接抛错
            return !obj.window //IE6-8 window
        }
    }
    return false
}

//avalon.mobile更倚重Object.prototoype.toString来判定
```

```
function isArrayLike(obj) {
    if (obj && typeof obj === 'object') {
        var n = obj.length,
                str = toString.call(obj)
        if (rarraylike.test(str)) {
            return true
        } else if (str === '[object Object]' && n === (n >>> 0)
) {
        //由于ecma262v5能修改对象属性的enumerable，因此不能用propert
yIsEnumerable来判定了
            return true
        }
    }
    return false
}
```

jQuery的isXML方法，有一段有趣的发展历程，这个将在第6章详解，这里略过。

avalon的isVML方法，也就avalon这种偏执于兼容性的框架，才会发掘出这样的函数。

```
function isVML(src) {
    var nodeName = src.nodeName
    return nodeName.toLowerCase() === nodeName && src.scopeName
 && src.outerText === ''
}
```

基于实用主义，我们有时不得不妥协。百度的

tangram就是典型，与EXT一样，能想到的都写上，而且判定非常严谨。

```
baidu.isDate = function(o) {
    return {}.toString.call(o) === "[object Date]" && o.toStrin
g() !== 'Invalid Date' && !isNaN(o);
}
baidu.isNumber = function(o) {
    return '[object Number]' == {}.toString.call(o) && isFinite
(o);
}
```

1.6　domReady

domReady其实是一种名为DOMContentLoaded事件的别称。不过由于框架的需要，它与真正的DOMContentLoaded有一点区别。在许多JavaScript书籍中，它们都会教导我们把JavaScript逻辑写在window.onload回调中，以防DOM树还没有建完就开始对节点进行操作，导致出错。而对于框架来说，越早介入对DOM的干涉就越好，例如要进行特征侦测之类的。domReady还可以满足用户提前绑定事件的需求。因为有时网页的图片等资源过多，window.onload就迟迟不能触发，这时若还没有绑定事件，用户点击哪个按钮都没反应（除了跳转页面）。因此主流框架都引入domReady机制，并且费了很大劲兼容所有浏览器，具体策略如下。

（1）对于支持DOMContentLoaded事件的使用DOMContentLoaded事件。

（2）旧版本IE使用Diego Perini发现的著名hack!

```
//http://javascript.nwbox.com/IEContentLoaded/
//by Diego Perini 2007.10.5
function IEContentLoaded(w, fn) {
    var d = w.document, done = false,
            init = function() {
        if (!done) {//只执行一次
            done = true;
            fn();
        }
    };
    (function() {
        try {//在DOM未建完之前调用元素doScroll抛出错误
            d.documentElement.doScroll('left');
        } catch (e) {//延迟再试
            setTimeout(arguments.callee, 50);
            return;
        }
        init();//没有错误则执行用户回调
    })();
    // 如果用户是在domReady之后绑定这个函数，则立即执行它
    d.onreadystatechange = function() {
        if (d.readyState == 'complete') {
            d.onreadystatechange = null;
            init();
        }
    };
}
```

此外，IE还可以通过script defer hack进行判定。

```
//http://webreflection.blogspot.com/search?q=onContent
//by Andrea Giammarchi 2006.9.24
document.write("<script id=__ie_onload defer src=//0><\/scr" +
"ipt>");
script = document.getElementById("__ie_onload");
script.onreadystatechange = function() {//IE即使是死链也能触发事件
    if (this.readyState == "complete"){
        init(); // 指定了defer的script会在DOM树建完才触发
    };
}
```

不过有个问题，如果我们的种子模块是动态加载的，在它插入DOM树时，DOM树已经建完了，这该怎么触发我们的ready回调呢？jQuery给出的方案是，onload也一起被监听。但是如果用户的脚本是onload之后才加载进来呢？那么只好判定一下**document.readyState** 是否等于**complete**，如果是，则说明页面早就domReady，可以执行用户的回调。不过，这个“保险丝”还是存在问题，因为Firefox 3.6没有**document.readyState** 属性！那么我们来看avalon给出的方案。

```
var readyList = [];
avalon.ready = function(fn) {
    if (readyList) {
        readyList.push(fn);
```

```
    } else {
        fn();
    }
}
var readyFn, ready = W3C ? "DOMContentLoaded" : "readystatechan
ge";
function fireReady() {
    for (var i = 0, fn; fn = readyList[i++]; ) {
        fn();
    }
    readyList = null;
    fireReady = avalon.noop; //惰性函数，防止IE9二次调用_checkDeps
}

function doScrollCheck() {
    try { //IE下通过doScrollCheck检测DOM树是否建完
        html.doScroll("left");
        fireReady();
    } catch (e) {
        setTimeout(doScrollCheck);
    }
}

//在Firefox 3.6之前，不存在readyState属性
//http://www.cnblogs.com/rubylouvre/archive/2012/12/18/2822912.
html
if (!DOC.readyState) {
    var readyState = DOC.readyState = DOC.body ? "complete" : "
loading";
}
if (DOC.readyState === "complete") {
    fireReady(); //如果在domReady之外加载
} else {
    avalon.bind(DOC, ready, readyFn = function() {
        if (W3C || DOC.readyState === "complete") {
            fireReady();
            if (readyState) { //IE下不能改写DOC.readyState
                DOC.readyState = "complete";
            }
        }
    });
    if (html.doScroll) {
        try { //如果跨域会报错，那时肯定证明是存在两个窗口的
            if (self.eval === parent.eval) {
                doScrollCheck();
```

```
            }
        } catch (e) {
            doScrollCheck();
        }
    }
}
```

1.7 无冲突处理

无冲突处理也称为多库共存。不得不说，$是最重要的函数名。许多框架都爱用它作为自己的命名空间。在jQuery还比较弱小时，如何让人们试用它呢？当时Prototype是主流，于是jQuery发明了noConflict函数，下面是源代码。

```
var
       window = this,
       undefined,
       _jQuery = window.jQuery,
       _$ = window.$,
       //把window存入闭包中的同名变量，方便内部函数在调用window时不用费
大力气查找它
       //_jQuery与_$用于以后重写
       jQuery = window.jQuery = window.$ = function(selector, c
ontext) {
    //用于返回一个jQuery对象
    return new jQuery.fn.init(selector, context);
}

jQuery.extend({
    noConflict: function(deep) {
        //引入jQuery类库后，闭包外面的window.$与window.jQuery都储存着
一个函数
        //它是用来生成jQuery对象或在domReady后执行其中的函数
        //回顾最上面的代码，在还没有把function赋给它们时，_jQuery与_$已
经被赋值了
        //因此它们俩的值必然是undefined
        //因此这种放弃控制权的技术很简单，就是用undefined把window.$里面
的jQuery系的函数清除
```

```
        //这时Prototype或mootools的$就可以了
        window.$ = _$;//相当于window.$ = undefined
        //这时就要为noConflict添加一个布尔值，为true
        if (deep)
            //但我们必须使用一个接纳jQuery对象与jQuery的入口函数
            //闭包里面的内容除非被window等宿主对象引用，否则就是不可见的
            //因此我们把闭包里面的jQuery return出去，外面用一个变量接纳
就可以
            window.jQuery = _jQuery;//相当window.jQuery = undefi
ned
        return jQuery;
    }
});
```

使用时，先引入其他的库，然后引入jQuery，调用$.noConflict()进行改名，这样就不影响其他人的$运作了。

1.8 总结

种子模块是一个非常核心的模块，其中包含了近20年来JavaScript积累的最常用的函数与功能，大家务必学好，这也是面试时的常考的内容。

---

[1] AMD全称是Asynchronous Module Definition，即异步模块加载机制。从它的规范描述页面看，AMD很短也很简单，但它却完整描述了模块的定义、依赖关系、引用关系以及加载机制。从它被RequireJS、Node.js、dojo、jQuery的使用中也可以看出它具有很大的价值。

[2] CommonJS规范是JavaScript在服务器端的规范，Node.js采用了这个规范。根据CommonJS规范，一个单独的文件就是一个模块。加载模块使用require方法。该方法读取一个文件并执行，最后返

回文件内部的exports对象。CommonJS构建的这套模块导出和引入机制使得用户完全不必考虑变量污染、命名空间等方案与之相比相形见绌。

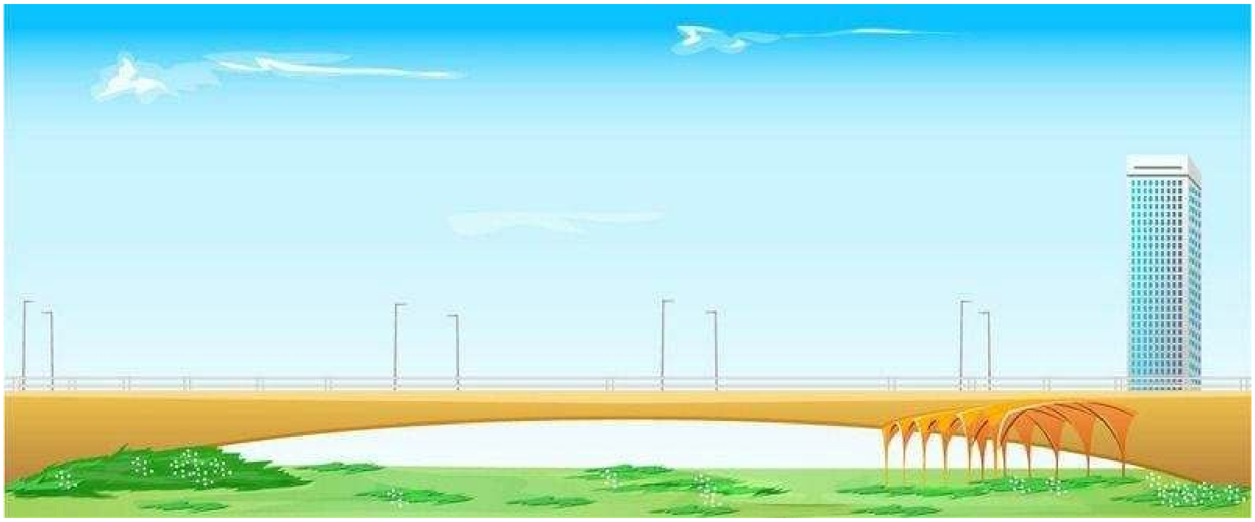
[3]　在ES5之前，JavaScript没有内置的机制来指定或者检查对象某个属性（property）的特性（characteristics），例如某个属性是只读（readonly）的或者不能被枚举（enumerable）的。但是在ES5之后，JavaScript被赋予了这个能力，所有的对象属性都可以通过属性描述符（Property Descriptor）来指定。

[4]　Polyfilling是由RemySharp提出的一个术语，它是用来描述复制缺少的API和API功能的行为。你可以使用它编写单独应用的代码，而不用担心其他浏览器原生是不是支持。实际上，polyfills并不是新技术，也不是和HTML5捆绑到一起的。我们已经在如json2.js、ie7-js和为IE浏览器提供透明PNG支

持的JS中使用过了。而和现在polyfills的区别就是新增的HTML5 polyfills。

[5] 我也写过一篇博文专门探索这问题 http://www.cnblogs.com/rubylouvre/archive/2010/02/2(

第2章　语言模块



1995年，Brendan Eich读完了在程序语言设计中曾经出现过的所有错误，自己又发现了一些更多的错误，然后用它们创造出了LiveScript。之后，为了紧跟Java语言的潮流，它被重新命名为JavaScript。再然后，为了追随一种皮肤病的时髦名字，这个语言又命名为ECMAScript。

上面一段话出自博文《编程语言伪简史》。可

见，JavaScript受到了多么辛辣的嘲讽，它在当时是多么不受欢迎。抛开偏见，JavaScript的确有许多不足之处。由于互联网的传播性及浏览器厂商大战，JavaScript之父失去了对此门语言的掌控权。即便他想修复这些bug或推出某些新特性，也要所有浏览器厂商都点头才行。IE6的市场独占性，打破了他的奢望。这个局面直到Chrome诞生，才有所改善。

但在IE6时期，浏览器提供的原生API数量是极其贫乏的，因此各个框架都创造了许多方法来弥补这缺陷。视框架作者原来的语言背景不同，这些方法也是林林总总。其中最杰出的代表是王者Prototype.js，把ruby语言的那一套方式或范式搬过来，从底层促进了JavaScript的发展。ECMA262V6添加那一堆字符串、数组方法，差不多就是改个名字而已。

即便是浏览器的API也不能尽信，尤其是IE6、

IE7、IE8到处是bug。早期出现的各种“JS库”，例如远古的prototype、中古的mootools，到近代的jQuery，再到大规模、紧封装的YUI和Extjs，很大的一个目标就是为了填“兼容性”这个“大坑”。

在avalon2中，就提供了许多带compact命名的模块，它们就是专门用于修复古老浏览器的兼容性问题。此外，本章也介绍了一些非常底层的知识点，能让读者更熟悉这门语言。

2.1　字符串的扩展与修复

笔者发现脚本语言都对字符串特别关注，有关它的方法特别多。笔者把这些方法分为三大类，如图2-1所示。

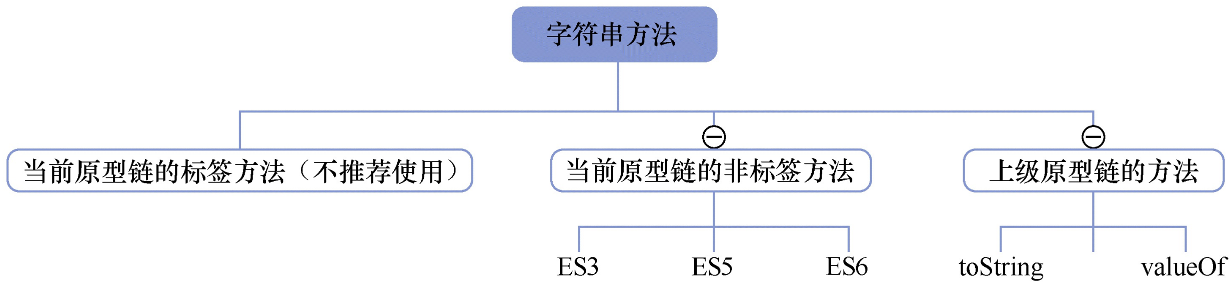


图2-1

显然以前，总是想着通过字符串生成标签，于是诞生了一些方法，如anchor、big、blink、bold、fixed、fontcolor、italics、link、small、strike、sub及sup。

剩下的就是charAt、charCodeAt、concat、indexOf、lastIndexOf、localeCompare、match、

replace，search、slice、split、substr、substring、toLocaleLowerCase、toLocaleUpperCase、toLowerCase、toUpperCase及从Object继承回来的方法，如toString、valueOf。

鲜为人知的是，数值的toString有一个参数，通过它可以转换为进行进制的数值，如图 2-2所示。

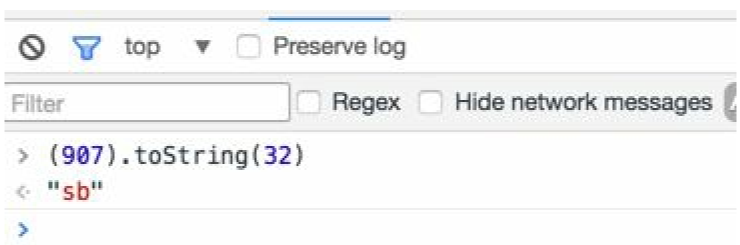


图2-2

但相对于其他语言，JavaScript的字符串方法可以说是十分贫乏的，因此后来的ES5、ES6又加上了一堆方法。

即便这样，也很难满足开发需求，比如说新增的方法就远水救不了近火。因此各大名库都提供了

一大堆操作字符串的方法。我综合一下Prototype、mootools、dojo、EXT、Tangram、RightJS的一些方法，进行比较去重，在mass Framework为字符串添加如下扩展：contains、startsWith、endsWith、repeat、camelize、underscored、capitalize、stripTags、stripScripts、escapeHTML、unescapeHTML、escapeRegExp、truncate、wbr、pad，写框架的读者可以视自己的情况进行增减，如图2-3所示。其中前4个是ECMA262V6的标准方法；接着9个发端于**Prototype.js** 广受欢迎的工具方法；wbr则来自**Tangram** ，用于软换行 ，这是出于汉语排版的需求。pad也是一个很常用的操作，已被收录，如图2-3所示。

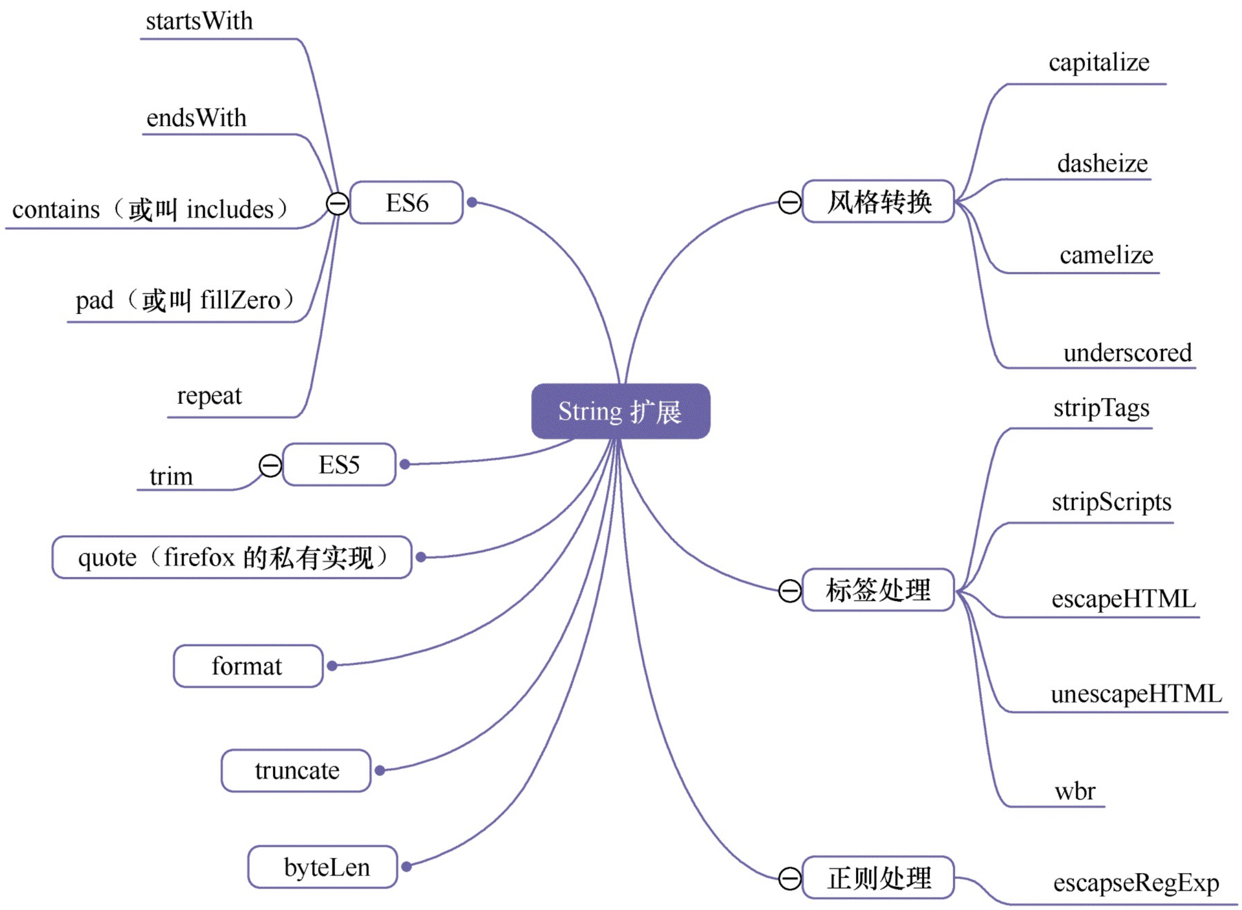


图2-3

到了另一个框架**avalon2**，笔者的方法也有用武之地，或者改成avalon的静态方法，或者作为ECMA262V6的补丁模块，或者作为过滤器（如camelize、truncate）。

各种方法实现如下。

contains 方法：判定一个字符串是否包含另一个字符串。常规思维是使用正则表达式。但每次都要用new RegExp来构造，性能太差，转而使用原生字符串方法，如indexOf、lastIndexOf、search。

```
function contains(target, it) {
   //indexOf改成search，lastIndexOf也行得通
   return target.indexOf(it) != -1;
}
```

在Mootools版本中，笔者看到它支持更多参数，估计目的是判定一个元素的className是否包含某个特定的class。众所周知，元素可以添加多个class，中间以空格隔开，使用mootools的contains就能很方便地检测包含关系了。

```
function contains(target, str, separator) {
    return separator ?
            (separator + target + separator).indexOf(separator
+ str + separator) > -1 :
            target.indexOf(str) > -1;
}
```

startsWith方法：判定目标字符串是否位于原字符串的开始之处，可以说是contains方法的变种。

```
//最后一个参数是忽略大小写
function startsWith(target, str, ignorecase) {
    var start_str = target.substr(0, str.length);
    return ignorecase ? start_str.toLowerCase() === str.toLower
Case() :
            start_str === str;
}
```

endsWith方法：与startsWith方法相反。

```
//最后一个参数是忽略大小写
function endsWith(target, str, ignorecase) {
    var end_str = target.substring(target.length - str.length);
    return ignorecase ? end_str.toLowerCase() === str.toLowerCa
se() :
            end_str === str;
}
```

2.1.1 repeat

repeat方法：将一个字符串重复自身N次，如repeat（"ruby", 2）得到rubyruby。

版本1：利用空数组的join方法。

```
function repeat(target, n) {
    return (new Array(n + 1)).join(target);
}
```

版本2：版本1的改良版。创建一个对象，使其拥有length属性，然后利用call方法去调用数组原型的join方法，省去创建数组这一步，性能大为提高。重复次数越多，两者对比越明显。另外，之所以要创建一个带length属性的对象，是因为要调用数组的原型方法，需要指定call的第一个参数为类数组对象，而类数组对象的必要条件是其length属性的值为非负整数。

```
function repeat(target, n) {
    return Array.prototype.join.call({
        length: n + 1
    }, target);
}
```

版本3：版本2的改良版。利用闭包将类数组对

象与数组原型的join方法缓存起来，避免每次都重复创建与寻找方法。

```
var repeat = (function() {
    var join = Array.prototype.join, obj = {};
    return function(target, n) {
        obj.length = n + 1;
        return join.call(obj, target);
    }
})();
```

版本 4：从算法上着手，使用二分法，比如我们将ruby重复5次，其实我们在第二次已得到rubyruby，那么第3次直接用rubyruby进行操作，而不是用ruby。

```
function repeat(target, n) {
    var s = target, total = [];
    while (n > 0) {
        if (n % 2 == 1)
            total[total.length] = s;//如果是奇数
        if (n == 1)
            break;
        s += s;
        n = n >> 1;//相当于将n除以2取其商,或说开2二次方
    }
    return total.join('');
}
```

版本5：版本4的变种，免去创建数组与使用jion方法。它的短处在于它在循环中创建的字符串比要求的还长，需要回减一下。

```
function repeat(target, n) {
    var s = target, c = s.length * n
    do {
        s += s;
    } while (n = n >> 1);
    s = s.substring(0, c);
    return s;
}
```

版本6：版本4的改良版。

```
function repeat(target, n) {
    var s = target, total = "";
    while (n > 0) {
        if (n % 2 == 1)
            total += s;
        if (n == 1)
            break;
        s += s;
        n = n >> 1;
    }
    return total;
}
```

版本7：与版本6相近。不过在浏览器下递归好

像都做了优化（包括IE6），与其他版本相比，属于上乘方案之一。

```
function repeat(target, n) {
    if (n == 1) {
        return target;
    }
    var s = repeat(target, Math.floor(n / 2));
    s += s;
    if (n % 2) {
        s += target;
    }
    return s;
}
```

版本8：可以说是一个反例，很慢，不过实际上它还是可行的，因为实际上没有人将n设成上百成千。

```
function repeat(target, n) {
    return (n <= 0) ? "" : target.concat(repeat(target, --n));
}
```

经测试，版本**6** 在各浏览器的得分是最高的。

2.1.2　byteLen

byteLen方法：取得一个字符串所有字节的长度。这是一个后端过来的方法，如果将一个英文字符插入数据库char、varchar、text类型的字段时占用一个字节，而将一个中文字符插入时占用两个字节。为了避免插入溢出，就需要事先判断字符串的字节长度。在前端，如果我们要用户填写文本，限制字节上的长短，比如发短信，也要用到此方法。随着浏览器普及对二进制的操作，该方法也越来越常用。

版本 1：假设当字符串每个字符的Unicode编码均小于或等于255时，byteLength为字符串长度；再遍历字符串，遇到Unicode编码大于255时，为byteLength补加1。

```
function byteLen(target) {
    var byteLength = target.length, i = 0;
    for (; i < target.length; i++) {
        if (target.charCodeAt(i) > 255) {
            byteLength++;
        }
    }
    return byteLength;
}
```

版本2：使用正则表达式，并支持设置汉字的存储字节数。比如用mysql存储汉字时，是3个字节数。

```
function byteLen(target, fix) {
    fix = fix ? fix : 2;
    var str = new Array(fix + 1).join("-")
    return target.replace(/[^\x00-\xff]/g, str).length;
}
```

版本3：来自腾讯的解决方案。腾讯通过多子域名+postMessage+manifest离线proxy页面的方式扩大localStorage的存储空间。在这个过程中，我们需要知道用户已经保存了多少内容，因此就必须编写一个严谨的byteLen方法。

```
/**
 * http://www.alloyteam.com/2013/12/js-calculate-the-number-of-bytes-occupied-by-a-string/
 * 计算字符串所占的内存字节数，默认使用UTF-8的编码方式计算，也可制定为UTF-16
 * UTF-8 是一种可变长度的 Unicode 编码格式，使用1～4个字节为每个字符编码
 *
 * 000000 - 00007F(128个代码)     0zzzzzzz(00-7F)
              1个字节
```

```
 * 000080 - 0007FF(1920个代码)    110yyyyy(C0-DF) 10zzzzzz(80-BF
)              2个字节
 * 000800 - 00D7FF
   00E000 - 00FFFF(61440个代码)   1110xxxx(E0-EF) 10yyyyyy 10zzz
zzz            3个字节
 * 010000 - 10FFFF(1048576个代码) 11110www(F0-F7) 10xxxxxx 10yyy
yyy 10zzzzzz  4个字节
 *
 * 注: Unicode在范围 D800-DFFF 中不存在任何字符
 * {@link <a onclick="javascript:pageTracker._trackPageview('/o
utgoing/zh.wikipedia. org/wiki/UTF-8');"
 * href="http://zh.wikipedia.org/wiki/UTF-8">http://zh.wikipedi
a.org/wiki/UTF-8</a>}
 *
 * UTF-16 大部分使用2个字节编码，编码超出 65535 的使用4个字节
 * 000000 - 00FFFF  2个字节
 * 010000 - 10FFFF  4个字节
 *
 * {@link <a onclick="javascript:pageTracker._trackPageview('/o
utgoing/zh.wikipedia. org/wiki/UTF-16');"
 * href="http://zh.wikipedia.org/wiki/UTF-16">http://zh.wikiped
ia.org/wiki/UTF-16</a>}
 * @param  {String} str
 * @param  {String} charset utf-8, utf-16
 * @return {Number}
 */
function byteLen(str, charset){
    var total = 0,
        charCode,
        i,
        len;
    charset = charset ? charset.toLowerCase() : '';
    if(charset === 'utf-16' || charset === 'utf16'){
        for(i = 0, len = str.length; i < len; i++){
            charCode = str.charCodeAt(i);
            if(charCode <= 0xffff){
                total += 2;
            }else{
                total += 4;
            }
        }
    }else{
        for(i = 0, len = str.length; i < len; i++){
            charCode = str.charCodeAt(i);
            if(charCode <= 0x007f) {
```

```
                total += 1;
            }else if(charCode <= 0x07ff){
                total += 2;
            }else if(charCode <= 0xffff){
                total += 3;
            }else{
                total += 4;
            }
        }
    }
    return total;
}
```

truncate方法：用于对字符串进行截断处理。当超过限定长度，默认添加3个点号。

```
function truncate(target, length, truncation) {
    length = length || 30;
    truncation = truncation === void(0) ? '...' : truncation;
    return target.length > length ?
            target.slice(0, length - truncation.length) + trunc
ation : String(target);
}
```

camelize方法：转换为驼峰风格。

```
function camelize(target) {
    if (target.indexOf('-') < 0 && target.indexOf('_') < 0) {
        return target;//提前判断，提高getStyle等的效率
    }
    return target.replace(/[-_][^-_]/g, function(match) {
        return match.charAt(1).toUpperCase();
    });
```

```
}
```

underscored方法：转换为下划线风格。

```
function underscored(target) {
    return target.replace(/([a-z\d])([A-Z])/g, '$1_$2').
            replace(/\-/g, '_').toLowerCase();
}
```

dasherize方法：转换为连字符风格，即CSS变量的风格。

```
function dasherize(target) {
    return underscored(target).replace(/_/g, '-');
}
```

capitalize方法：首字母大写。

```
function capitalize(target) {
    return target.charAt(0).toUpperCase() + target.substring(1)
.toLowerCase();
}
```

stripTags 方法：移除字符串中的html标签。

比如，我们需要实现一个HTMLParser，这时就要处理option元素的innerText问题。此元素的内部只能接受文本节点，如果用户在里面添加了span、strong等标签，我们就需要用此方法将这些标签移除。在Prototype.js中，它与strip、stripScripts是一组方法。

```
var rtag = /<\w+(\s+("[^"]*"|'[^']*'|[^>])+)?>|<\/\w+>/gi
function stripTags(target) {
    return String(target || "").replace(rtag, '');
}
```

stripScripts 方法：移除字符串中所有的script标签。弥补stripTags方法的缺陷。此方法应在stripTags之前调用。

```
function stripScripts(target) {
    return String(target || "").replace(/<script[^>]*>([\S\s]*?
)<\/script>/img, '')
}
```

escapeHTML 方法：将字符串经过html转义得

到适合在页面中显示的内容，如将“< ”替换为“<”`。此方法用于防止XSS攻击。

```
    function escapeHTML(target) {
    return target.replace(/&/g, '&amp;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;')
            .replace(/"/g, "&quot;")
            .replace(/'/g, "&#39;");
}
```

unescapeHTML方法：将字符串中的html实体字符还原为对应字符。

```
function unescapeHTML(target) {
    return String(target)
    .replace(/&#39;/g, '\'')
    .replace(/&quot;/g, '"')
    .replace(/&lt;/g, '<')
    .replace(/&gt;/g, '>')
    .replace(/&amp;/g, '&')
}
```

注意一下escapeHTML和unescapeHTML这两个方法，它们不但在replace的参数是反过来的，replace的顺序也是反过来的。它们在做html parser

非常有用的。但涉及浏览器，兼容性问题就一定会存在。

在citojs这个库中，有一个类似于escapeHTML的方法叫escapeContent，它是这样写的。

```
function escapeContent(value) {
        value = '' + value;
        if (isWebKit) {
            helperDiv.innerText = value;
            value = helperDiv.innerHTML;
        } else if (isFirefox) {
            value = value.split('&').join('&amp;').split('<').join('&lt;').split('>'). join('&gt;');
        } else {
            value = value.replace(/&/g, '&amp;').replace(/</g, '&lt;').replace(/>/g, '&gt;');
        }
        return value;
    }
```

看情况是处理`&`时出了分歧。但它们这么做其实也不能处理所有html实体。因此Prototype.js是建议使用原生API `innerHTML` , `innerText` 来处理。

```
var div = document.createElement('div')
```

```
var escapeHTML = function (a) {
    div.data = a
    return div.innerHTML
}

var unescapeHTML = function (a) {
    div.innerHTML = a
    return getText(div)//相当于innerText, textContent
}

function getText(node) {
    if (node.nodeType !== 1) {
        return node.nodeValue
    } else if (node.nodeName !== 'SCRIPT') {
        var ret = ''
        for (var i = 0, el; el = node.childNodes[i++]; ) {
            ret += getText(el)
        }
    } else {
        return ''
    }
}
```

但这样一来，它们就不能运行于Node.js环境中，并且性能也不好，于是人们发展出下面这些库。

```
https://github.com/mathiasbynens/he
https://github.com/mdevils/node-html-entities
```

escapeRegExp方法：将字符串安全格式化为

正则表达式的源码。

```
function escapeRegExp(target) {
    return target.replace(/([-.*+?^${}()|[\]\/\\])/g, '\\$1');
}
```

2.1.3 pad

pad方法：与**trim**方法 相反，pad可以为字符串的某一端添加字符串。常见的用法如日历在月份前补零，因此也被称之为fillZero。笔者在博客上收集许多版本的实现，在这里转换为静态方法一并写出。

版本1：数组法，创建数组来放置填充物，然后再在右边起截取。

```
function pad(target, n) {
    var zero = new Array(n).join('0');
    var str = zero + target;
    var result = str.substr(-n);
    return result;
}
```

版本2：版本1的变种。

```
function pad(target, n) {
    return Array((n + 1) - target.toString().split('').length).
join('0') + target;
}
```

版本3：二进制法。前半部分是创建一个含有n个零的大数，如（1<<5）.toString（2），生成100000，（1<<8）.toString（2）生成100000000，然后再截短。

```
function pad(target, n) {
    return (Math.pow(10, n) + "" + target).slice(-n);
}
```

版本4：Math.pow法，思路同版本3。

```
function pad(target, n) {
    return ((1 << n).toString(2) + target).slice(-n);
}
```

版本5：toFixed法，思路与版本3差不多，创建

一个拥有n个零的小数，然后再截短。

```
function pad(target, n) {
    return (0..toFixed(n) + target).slice(-n);
}
```

版本6：创建一个超大数，在常规情况下是截不完的。

```
function pad(target, n) {
    return (1e20 + "" + target).slice(-n);
}
```

版本7：质朴长存法，就是先求得长度，然后一个个地往左边补零，加到长度为n为止。

```
function pad(target, n) {
    var len = target.toString().length;
    while (len < n) {
        target = "0" + target;
        len++;
    }
    return target;
}
```

版本8：也就是现在mass Framework使用的版本，支持更多的参数，允许从左或从右填充，以及使用什么内容进行填充。

```
function pad(target, n, filling, right, radix) {
    var num = target.toString(radix || 10);
    filling = filling || "0";
    while (num.length < n) {
        if (!right) {
            num = filling + num;
        } else {
            num += filling;
        }
    }
    return num;
}
```

在ECMA262V7规范中，pad方法也有了对应的代替品——**padStart**，此外，还有从后面补零的方法——**padEnd**。

```
https://github.com/es-shims/es7-shim
```

wbr方法：为目标字符串添加wbr软换行。不过需要注意的是，它并不是在每个字符之后都插

入<wbr>字样，而是相当于在组成文本节点的部分中的每个字符后插入<wbr>字样。例如，aabbcc，返回a<wbr>a<wbr>b<wbr>b<wbr>c<wbr>c<wbr>。另外，在Opera下，浏览器默认css不会为wbr加上样式，导致没有换行效果，可以在css中加上wbr : after { content: "\00200B" }解决此问题。

```
function wbr(target) {
    return String(target)
            .replace(/(?:<[^>]+>)|(?:&#?[0-9a-z]{2,6};)|(.{1})/
gi, '$&<wbr>')
            .replace(/><wbr>/g, '>');
}
```

format方法：在C语言中，有一个叫printf的方法，我们可以在后面添加不同类型的参数嵌入到将要输出的字符串中。这是非常有用的方法，因为JavaScript涉及大量的字符串拼接工作。如果涉及逻辑，我们可以用模板；如果轻量点，我们可以用这个方法。它在不同框架中名字是不同的，

Prototype.js叫interpolate；Base2叫**format**；mootools叫**substitute**。

```
function format(str, object) {
    var array = Array.prototype.slice.call(arguments, 1);
    return str.replace(/\\?\#{([^{}]+)\}/gm, function(match, na
me) {
        if (match.charAt(0) == '\\')
            return match.slice(1);
        var index = Number(name)
        if (index >= 0)
            return array[index];
        if (object && object[name] !== void 0)
            return  object[name];
        return  '';
    });
}
```

format方法支持两种传参方法，如果字符串的占位符为0、1、2这样的非零整数形式，要求传入两个或两个以上的参数，否则就传入一个对象，键名为占位符。

```
var a = format("Result is #{0},#{1}", 22, 33);
alert(a);//"Result is 22,33"
var b = format("#{name} is a #{sex}", {
    name: "Jhon",
    sex: "man"
});
alert(b);//"Jhon is a man"
```

2.1.4 quote

quote 方法：在字符串两端添加双引号，然后内部需要转义的地方都要转义，用于接装JSON的键名或模板系统中。

版本1：来自JSON3。

```
//avalon2
//https://github.com/bestiejs/json3/blob/master/lib/json3.js
var Escapes = {
    92: "\\\\",
    34: '\\"',
    8: "\\b",
    12: "\\f",
    10: "\\n",
    13: "\\r",
    9: "\\t"
}

// Internal: Converts 'value' into a zero-padded string such th
at its
// length is at least equal to 'width'. The 'width' must be <=
6.
var leadingZeroes = "000000"
var toPaddedString = function (width, value) {
    // The '|| 0' expression is necessary to work around a bug
in
    // Opera <= 7.54u2 where '0 == -0', but 'String(-0) !== "0"
'.
    return (leadingZeroes + (value || 0)).slice(-width)
};
var unicodePrefix = "\\u00"
```

```
var escapeChar = function (character) {
    var charCode = character.charCodeAt(0), escaped = Escapes[c
harCode]
    if (escaped) {
        return escaped
    }
    return unicodePrefix + toPaddedString(2, charCode.toString(
16))
};
var reEscape = /[\x00-\x1f\x22\x5c]/g
function quote(value) {
    reEscape.lastIndex = 0
    return '"' + ( reEscape.test(value)? String(value).replace(
reEscape, escapeChar) : value ) + '"'
}

avalon.quote = typeof JSON !== 'undefined' ? JSON.stringify : q
uote
```

版本2：来自百度的etpl模板库。

```
//https://github.com/ecomfe/etpl/blob/2.1.0/src/main.js#L207
function stringLiteralize(source) {
    return '"'
            + source
            .replace(/\x5C/g, '\\\\')
            .replace(/"/g, '\\"')
            .replace(/\x0A/g, '\\n')
            .replace(/\x09/g, '\\t')
            .replace(/\x0D/g, '\\r')
            + '"';
}
```

当然，如果浏览器已经支持原生JSON，我们

直接用JSON.stringify就行了。另外，FF在JSON发明之前，就支持String.prototype.quote与String.quote方法，我们在使用quote之前需要判定浏览器是否内置这些方法。

接下来，我们来修复字符串的一些bug。字符串相对其他基础类型，没有太多bug，主要是3个问题。

（1）IE6、IE7不支持用数组中括号取它的每一个字符，需要用charAt来取。

（2）IE6、IE7、IE8不支持垂直分表符，于是诞生了`var isIE678= !+"\v1"`这个伟大的判定hack。

（3）IE对空白的理解与其他浏览器不一样，因此实现**trim**方法 会有一些不同。

前两个问题只能回避，我们重点研究第3个问题，也就是如何实现trim方法。由于太常用，所以相应的实现也非常多。我们可以一起看看，顺便学习一下正则。

2.1.5　trim与空白

版本1：虽然看起来不怎么样，但是动用了两次正则替换，实际速度非常惊人，这主要得益于浏览器的内部优化。base2类库使用这种实现。在Chrome刚出来的年代，这实现是异常快的，但chrome对字符串方法的疯狂优化，引起了其他浏览器的跟风。于是正则的实现再也比不了字符串方法了。一个著名的字符串拼接例子，直接相加比用Array做成的StringBuffer还快，而StringBuffer技术在早些年备受推崇！

```
function trim(str) {
    return str.replace(/^\s\s*/, '').replace(/\s\s*$/, '');
}
……
```

版本2：和版本1很相似，但稍慢一点，主要原因是它最先是假设至少存在一个空白符。Prototype.js使用这种实现，不过其名字为strip，因为Prototype的方法都是力求与Ruby同名。

```
<div class="se-preview-section-delimiter"></div>

…javascript
function trim(str) {
    return str.replace(/^\s+/, '').replace(/\s+$/, '');
}
```

版本 3：截取方式取得空白部分（当然允许中间存在空白符），总共调用了 4 个原生方法。设计非常巧妙，substring以两个数字作为参数。Math.max以两个数字作参数，search则返回一个数字。速度比上面两个慢一点，但基本比10之前的版本快！

```
function trim(str) {
    return str.substring(Math.max(str.search(/\S/), 0),
            str.search(/\S\s*$/) + 1);
}
```

版本4：这个可以称得上版本2的简化版，就是利用候选操作符连接两个正则。但这样做就失去了浏览器优化的机会，比不上版本3。由于看来很优雅，许多类库都使用它，如jQuery与Mootools。

```
function trim (str) {
    return str.replace(/^\s+|\s+$/g, '');
}
```

版本 5：match 如果能匹配到东西会返回一个类数组对象，原字符匹配部分与分组将成为它的元素。为了防止字符串中间的空白符被排除，我们需要动用到非捕获性分组（?:exp）。由于数组可能为空，我们在后面还要做进一步的判定。好像浏览器在处理分组上比较无力，一个字慢。所以不要迷信正则，虽然它基本上是万能的。

```
function trim(str) {
    str = str.match(/\S+(?:\s+\S+)*/);
    return str ? str[0] : '';
```

```
}
```

版本6：把符合要求的部分提供出来，放到一个空字符串中。不过效率很差，尤其是在IE6中。

```
function trim(str) {
    return str.replace(/^\s*(\S*(\s+\S+)*)\s*$/, '$1');
}
```

版本7：与版本6很相似，但用了非捕获分组进行了优点，性能较之有一点点提升。

```
function trim(str) {
    return str.replace(/^\s*(\S*(?:\s+\S+)*)\s*$/, '$1');
}
```

版本8：沿着上面两个的思路进行改进，动用了非捕获分组与字符集合，用“?”顶替了“*”，效果非常惊人。尤其在IE6中，可以用疯狂来形容这次性能的提升，直接秒杀FF3。

```
function trim(str) {
```

```
    return str.replace(/^\s*((?:[\S\s]*\S)?)\s*$/, '$1');
}
```

版本9：这次是用懒惰匹配顶替非捕获分组，在火狐中得到改善，IE没有上次那么疯狂。

```
function trim(str) {
    return str.replace(/^\s*([\S\s]*?)\s*$/, '$1');
}
```

版本 10：笔者只想说，搞出这个的人已经不能用厉害来形容，而是专家级别了。它先是把可能的空白符全部列出来，在第一次遍历中砍掉前面的空白，第二次砍掉后面的空白。全过程只用了indexOf与substring这个专门为处理字符串而生的原生方法，没有使用到正则。速度快得惊人，估计直逼内部的二进制实现，并且在IE与火狐（其他浏览器当然也毫无疑问）都有良好的表现，速度都是零毫秒级别的，PHP.js就收纳了这个方法。

```
Function trim(str) {
```

```
var whitespace = ' \n\r\t\f\x0b\xa0\u2000\u2001\u2002\u2003\n\
  \u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\
u3000';
for (var I = 0; I < str.length; I++) {
        if (whitespace.indexOf(str.charAt(i)) === -1) {
str = str.substring(i);
break;
        }
}
for (I = str.length – 1; I >= 0; I--) {
        if (whitespace.indexOf(str.charAt(i)) === -1) {
str = str.substring(0, I + 1);
break;
        }
}
return whitespace.indexOf(str.charAt(0)) === -1 ? str : ‘’;
}
```

版本 11：实现10的字数压缩版，前面部分的空白由正则替换负责砍掉，后面用原生方法处理，效果不逊于原版，但速度都非常逆天。

```
Function trim(str) {
str = str.replace(/^\s+/, '');
for (var I = str.length – 1; I >= 0; I--) {
        if (/\S/.test(str.charAt(i))) {
str = str.substring(0, I + 1);
break;
        }
}
return str;
}
```

版本12：版本10更好的改进版，注意说的不是性能速度，而是易记与使用方面。

```
Function trim(str) {
var m = str.length;
for (var I = -1; str.charCodeAt(++I) <= 32; )
for (var j = m – 1; j > I && str.charCodeAt(j) <= 32; j--)
return str.slice(I, j + 1);
}
```

但这还没有完。如果你经常翻看jQuery的实现，你就会发现jQuery1.4之后的trim实现，多出了一个对xA0的特别处理。这是Prototype.js的核心成员·kangax的发现，IE或早期的标准浏览器在字符串的处理上都有bug，把许多本属于空白的字符没有列为\s，jQuery在1.42中也不过把常见的不断行空白xA0修复掉，并不完整，因此最佳方案还是版本10。

```
// Make sure we trim BOM and NBSP
var rtrim = /^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g,
jQuery.trim = function( text ) {
        return text == null ?
"" :
( text + "" ).replace( rtrim, "" );
```

```
}
```

下面是一个比较晦涩的知识点——空白字符。根据屈屈的博文[1]，浏览器会把WhiteSpace和LineTerminator都列入空白字符。Ecma262 v5文档规定的WhiteSpace，如表2-1所示。

表**2-1**

Unicode编码	说明
U+0020	" " "\x20", "\u0020", <SP>半角空格符，键盘空格键
U+0009	"\t", "\x09", "\u0009", <TAB>制表符，键盘tab键
U+000B	"\v", "\x0B", "\u000B",<VT>垂直制表符
U+000C	"\f", "\x0C", "\u000C",<FF>换页符
U+000D	"\r", "\x0D", "\u000D",<CR>回车符
U+000A	"\n", "\x0A", "\u000A",<LF>换行符
U+00A0	"\xA0", "\u00A0",<NBSP>禁止自动换行空格符

U+1680	OGHAM SPACE MARK，欧甘空格
U+180E	Mongolian Vowel Separator，蒙古文元音分隔符
U+2000	EN QUAD
U+2001	EM QUAD
U+2002	EN SPACE，En空格。与En同宽（Em的1/2）
U+2003	EM SPACE，Em空格。与Em同宽
U+2004	THREE-PER-EM SPACE，Em 1/3空格
U+2005	FOUR-PER-EM SPACE，Em 1/4空格
U+2006	SIX-PER-EM SPACE，Em 1/6空格
U+2007	FIGURE SPACE，数字空格。与单一数字同宽
U+2008	PUNCTUATION SPACE，标点空格。与同字体窄标点同宽
U+2009	THIN SPACE，窄空格。Em 1/6或1/5宽
U+200A	HAIR SPACE，更窄空格。比窄空格更窄
U+200B	Zero Width Space，<ZWSP>，零宽空格

U+200C	Zero Width Non Joiner，<ZWNJ>，零宽不连字空格
U+200D	Zero Width Joiner，<ZWJ>，零宽连字空格
U+202F	NARROW NO-BREAK SPACE，窄式不换行空格
U+2028	<LS>行分隔符
U+2029	<PS>段落分隔符
U+205F	中数学空格。用于数学方程式
U+2060	Word Joiner，同U+200B，但该处不换行。Unicode 3.2新增，代替U+FEFF
U+3000	IDEOGRAPHIC SPACE，<CJK>，表意文字空格，即全角空格
U+FEFF	Byte Order Mark，<BOM>，字节次序标记字符。不换行功能于Unicode 3.2起废止

2.2 数组的扩展与修复

得益于Prototype.js的ruby式数组方法的侵略，让Jser()前端工程师大开眼界，原来对数组的操作也如此丰富多彩。原来JavaScript的数组方法就是基于栈与队列的那一套，像splice还是很晚加入的。让我们回顾一下它们的用法，如图2-4所示。

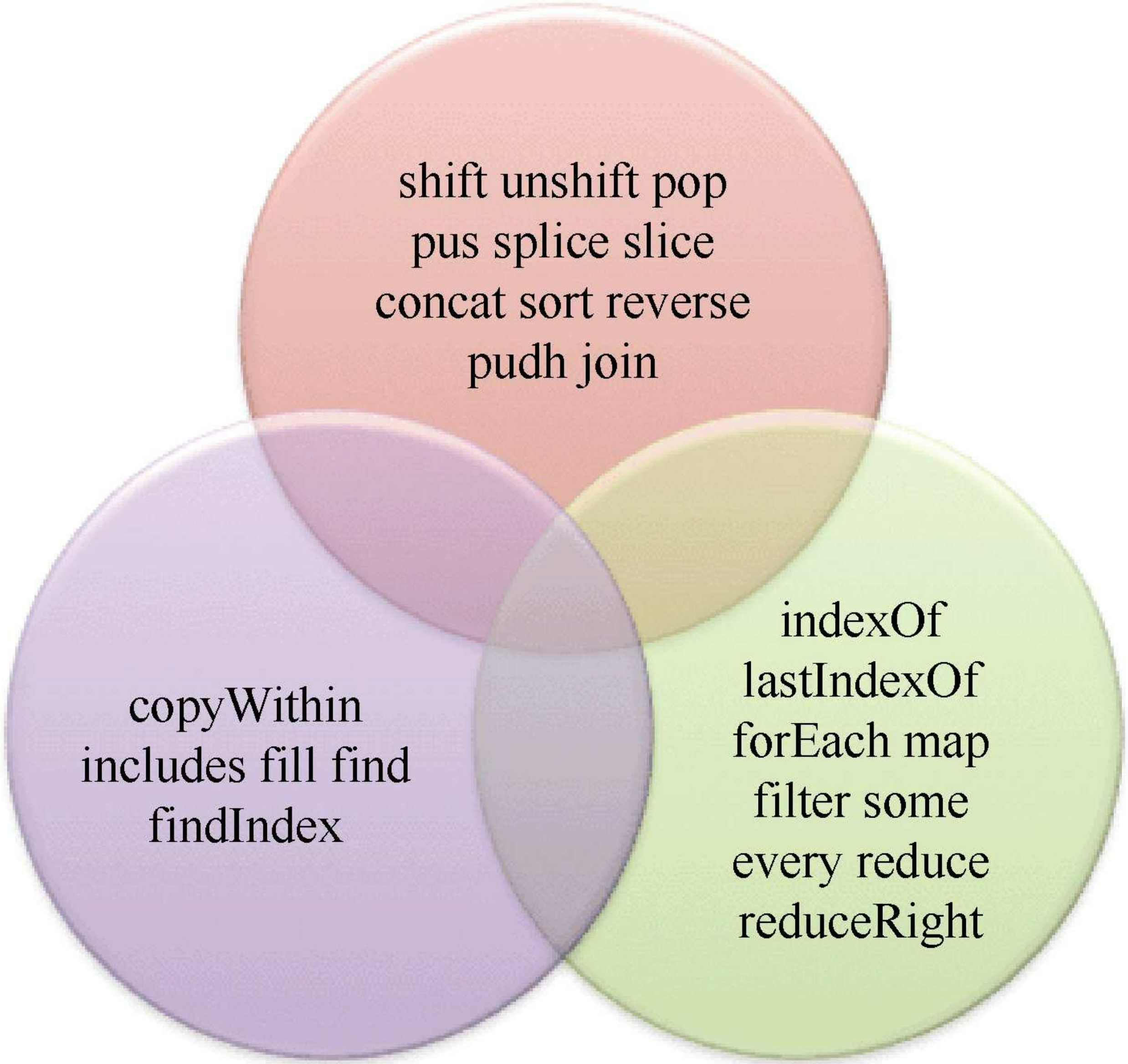


图2-4

- pop方法：出栈操作，删除并返回数组的最后一个元素。
- push方法：入栈操作，向数组的末尾添加一个

或更多元素，并返回新的长度。

- shift方法：出队操作，删除并返回数组的第一个元素。
- unshift方法：入队操作，向数组的开头添加一个或更多元素，并返回新的长度。
- slice方法：切片操作，从数组中分离出一个子数组，功能类似于字符串的。

substring、slice和substr是“三兄弟”，常用于转换类数组对象为真正的数组。

- sort方法：对数组的元素进行排序，有一个可选参数，为比较函数。
- reverse方法：颠倒数组中元素的顺序。
- splice方法：可以同时用于原数组的增删操作，数组的remove方法就是基于它写成的。
- concat方法：用于把原数组与参数合并成一个新数组，如果参数为数组，那么它会把其第一维

的元素放入新数组中。因此我们可以利用它实现数组的平坦化操作与克隆操作。

- join方法：把数组的所有元素放入一个字符串，元素通过指定的分隔符进行分隔。你可以想象成字符串split的反操作。
- indexOf方法：定位操作，返回数组中第一个等于给定参数的元素的索引值。
- lastIndexOf方法：定位操作，同上，不过是从后遍历。索引操作可以说是字符串同名方法的翻版，存在就返回非负整数，不存在就返回−1。
- forEach方法：迭代操作，将数组的元素依次传入一个函数中执行。Ptototype.js中对应的名字为each。
- map方法：收集操作，将数组的元素依次传入一个函数中执行，然后把它们的返回值组成一个新数组返回。Ptototype.js中对应的名字为collect。
- filter方法：过滤操作，将数组的元素依次传入

一个函数中执行，然后把返回值为true的那个元素放入新数组返回。在Prototype.js中，它有3个名字，即select、filter和findAll。

- some方法：只要数组中有一个元素满足条件（放进给定函数返回true），那么它就返回true。Ptototype.js中对应的名字为any。
- every方法：只有数组中所有元素都满足条件（放进给定函数返回true），它才返回true。Ptototype.js中对应的名字为all。
- reduce方法：归化操作，将数组中的元素归化为一个简单的数值。Ptototype.js中对应的名字为inject。
- reduceRight方法：归化操作，同上，不过是从后遍历。

为了方便大家记忆，我们可以用图2-5搞懂数组的18种操作。

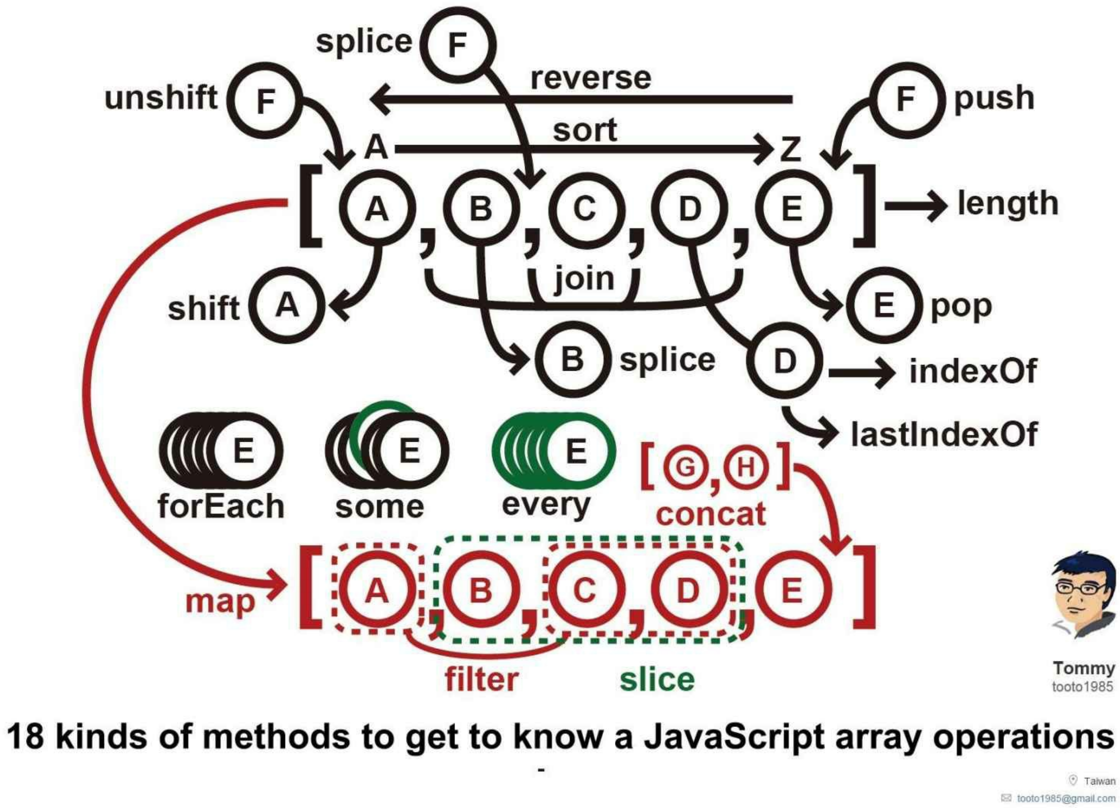


图2-5

由于许多扩展也基于这些新的标准化方法，因此笔者先给出IE6、IE7、IE8的兼容方案，全部在数组原型上修复它们。

```
[1, 2, , 4].forEach(function(e){
    console.log(e)
});
//依次打印出1，2，4，忽略第2、第3个逗号间的空元素
```

reduce与reduceRight是一组，我们可以利用reduce方法创建reduceRight方法。

```
ap.reduce = function(fn, lastResult, scope) {
    if (this.length == 0)
        return lastResult;
    var i = lastResult !== undefined ? 0 : 1;
    var result = lastResult !== undefined ? lastResult : this[0
];
    for (var n = this.length; i < n; i++)
        result = fn.call(scope, result, this[i], i, this);
    return result;
}

ap.reduceRight = function(fn, lastResult, scope) {
    var array = this.concat().reverse();
    return array.reduce(fn, lastResult, scope);
}
```

接下来，我们看看主流库为数组增加了哪些扩展吧。

Prototype.js的数组扩展：eachSlice、detect、grep、include、inGroupsOf、invoke、max、min、partition、pluck、reject、sortBy、zip、size、clear、first、last、compact、flatten、without、uniq、intersect、clone、inspect。

Rightjs的数组扩展：include、clean、clone、compact、empty、first、flatten、includes、last、max、merge、min、random、reject、shuffle、size、sortBy、sum、uniq、walk、without。

mootools的数组扩展：clean、invoke、associate、link、contains、append、getLast、getRandom、include、combine、erase、empty、flatten、pick、hexToRgb、rgbToHex。

EXT的数组扩展：contains、pluck、clean、unique、from、remove、include、clone、merge、intersect、difference、flatten、min、max、mean、sum、erase、insert。

Underscore.js的数组扩展：detect、reject、invoke、pluck、sortBy、groupBy、sortedIndex、first、last、compact、flatten、without、union、intersection、difference、uniq、zip。

qooxdoo的数组扩展：insertAfter、insertAt、insertBefore、max、min、remove、removeAll、removeAt、sum、unique。

Tangram的数组扩展：contains、empty、find、remove、removeAt、unique。

我们可以发现，Prototype.js那一套方法影响深远，许多库都有它的影子，全面而细节地囊括了各种操作，大家可以根据自己的需要与框架宗旨制订自己的数组扩展。笔者在这方面的考量如下，至少要包含平坦化、去重、乱序、移除这几个操作，其次是两个集合间的操作，如取并集、差集、交集。

下面是各种具体实现。

contains方法：判定数组是否包含指定目标。

```
function contains(target, item) {
    return target.indexOf(item) > -1
}
```

removeAt方法：移除数组中指定位置的元素，返回布尔值表示成功与否。

```
function removeAt(target, index) {
    return !!target.splice(index, 1).length
}
```

remove方法：移除数组中第一个匹配传参的那个元素，返回布尔值表示成功与否。

```
function remove(target, item) {
    var index = target.indexOf(item);
    if (~index)
        return removeAt(target, index);
    return false;
}
```

shuffle 方法：对数组进行洗牌。若不想影响原数组，可以先复制一份出来操作。有关洗牌算法的介绍，可见下面两篇博文。

《Fisher-Yates Shuffle》

《数组的完全随机排列》

```
function shuffle(target) {
    var j, x, i = target.length;
    for (; i > 0; j = parseInt(Math.random() * i),
            x = target[--i], target[i] = target[j], target[j] =
 x) {
    }
    return target;
}
```

random方法：从数组中随机抽选一个元素出来。

```
function random(target) {
    return  target[Math.floor(Math.random() * target.length)];
}
```

flatten方法：对数组进行平坦化处理，返回一个一维的新数组。

```
function flatten(target) {
    var result = [];
    target.forEach(function(item) {
        if (Array.isArray(item)) {
            result = result.concat(flatten(item));
        } else {
            result.push(item);
        }
```

```
    });
    return result;
}
```

unique方法：对数组进行去重操作，返回一个没有重复元素的新数组。

```
function unique(target) {
    var result = [];
    loop:  for (var i = 0, n = target.length; i < n; i++) {
        for (var x = i + 1; x < n; x++) {
            if (target[x] === target[i])
                continue loop;
        }
        result.push(target[i]);
    }
    return result;
}
```

compact方法：过滤数组中的null与undefined，但不影响原数组。

```
function compact(target) {
    return target.filter(function(el) {
        return el != null;
    });
}
```

pluck方法：取得对象数组的每个元素的指定属性，组成数组返回。

```
function pluck(target, name) {
    var result = [], prop;
    target.forEach(function(item) {
        prop = item[name];
        if (prop != null)
            result.push(prop);
    });
    return result;
}
```

groupBy方法：根据指定条件（如回调对象的某个属性）进行分组，构成对象返回。

```
function groupBy(target, val) {
    var result = {};
    var iterator = $.isFunction(val) ? val : function(obj) {
        return obj[val];
    };
    target.forEach(function(value, index) {
        var key = iterator(value, index);
        (result[key] || (result[key] = [])).push(value);
    });
    return result;
}
```

sortBy方法：根据指定条件进行排序，通常用

于对象数组。

```
function sortBy(target, fn, scope) {
    var array = target.map(function(item, index) {
        return {
            el: item,
            re: fn.call(scope, item, index)
        };
    }).sort(function(left, right) {
        var a = left.re, b = right.re;
        return a < b ? -1 : a > b ? 1 : 0;
    });
    return pluck(array, 'el');
}
```

union方法：对两个数组取并集。

```
function union(target, array) {
    return unique(target.concat(array));
}
```

intersect方法：对两个数组取交集。

```
function intersect(target, array) {
    return target.filter(function(n) {
        return ~array.indexOf(n);
    });
}
```

diff方法：对两个数组取差集（补集）。

```
function diff(target, array) {
    var result = target.slice();
    for (var i = 0; i < result.length; i++) {
        for (var j = 0; j < array.length; j++) {
            if (result[i] === array[j]) {
                result.splice(i, 1);
                i--;
                break;
            }
        }
    }
    return result;
}
```

min方法：返回数组中的最小值，用于数字数组。

```
function min(target) {
    return Math.min.apply(0, target);
}
```

max方法：返回数组中的最大值，用于数字数组。

```
function max(target) {
    return Math.max.apply(0, target);
}
```

基本上就这么多了，如果你想实现sum方法，可以使用reduce方法。我们再来抹平Array原生方法在各浏览器的差异，一个是IE6、IE7下unshift不返回数组长度的问题，一个splice的参数问题。unshift的bug很容易修复，可以使用函数劫持方式搞定。

```
if ([].unshift(1) !== 1) {
    var _unshift = Array.prototype.unshift;
    Array.prototype.unshift = function() {
        _unshift.apply(this, arguments);
        return this.length; //返回新数组的长度
    }
}
```

splice在一个参数的情况下，IE6、IE7、IE8默认第二个参数为零，其他浏览器为数组的长度，当然我们要以标准浏览器为准！

下面是最简单的修复方法。

```
if ([1, 2, 3].splice(1).length == 0) {
//如果是IE6、IE7、IE8，则一个元素也没有删除
    var _splice = Array.prototype.splice;
```

```
    Array.prototype.splice = function(a) {
        if (arguments.length == 1) {
            return _splice.call(this, a, this.length)
        } else {
            return _splice.apply(this, arguments)
        }
    }
}
```

下面是不利用任何原生方法的修复方法。

```
Array.prototype.splice = function(s, d) {
    var max = Math.max, min = Math.min,
            a = [], i = max(arguments.length - 2, 0),
            k = 0, l = this.length, e, n, v, x;
    s = s || 0;
    if (s < 0) {
        s += l;
    }
    s = max(min(s, l), 0);
    d = max(min(isNumber(d) ? d : l, l - s), 0);
    v = i - d;
    n = l + v;
    while (k < d) {
        e = this[s + k];
        if (e !== void 0) {
            a[k] = e;
        }
        k += 1;
    }
    x = l - s - d;
    if (v < 0) {
        k = s + i;
        while (x) {
            this[k] = this[k - v];
            k += 1;
            x -= 1;
        }
        this.length = n;
    } else if (v > 0) {
```

```
        k = 1;
        while (x) {
            this[n - k] = this[l - k];
            k += 1;
            x -= 1;
        }
    }
    for (k = 0; k < i; ++k) {
        this[s + k] = arguments[k + 2];
    }
    return a;
}
```

一旦有了splice方法，我们也可以自行实现pop、push、shift、unshift方法，因此你明白为什么这几个方法是直接修改原数组了吧？浏览器厂商的思路与我们一样，大概也是用splice方法来实现它们！

```
var ap = Array.prototype
var _slice = sp.slice;
ap.pop = function() {
    return this.splice(this.length - 1, 1)[0];
}

ap.push = function() {
    this.splice.apply(this,
            [this.length, 0].concat(_slice.call(arguments)));
    return this.length;
}

ap.shift = function() {
    return this.splice(0, 1)[0];
}
```

```
ap.unshift = function() {
    this.splice.apply(this,
            [0, 0].concat(_slice.call(arguments)));
    return this.length;
}
```

数组的空位

上面是一个forEach例子的演示，实质上我们通过修复原型方法的手段很难达到ecmascript规范的效果。缘故在于数组的空位，它在JavaScript的各个版本中都不一致。

数组的空位 是指数组的某一个位置没有任何值。比如，Array构造函数返回的数组都是空位。

```
Array(3) // [, , ,]
```

上面的代码中，Array(3)返回一个具有3个空位的数组。

注意，空位不是undefined，而是一个位置的值等于undefined，但依然是有值的。空位是没有任何值，in运算符可以说明这一点。

```
0 in [undefined, undefined, undefined] // true
0 in [, , ,] // false
```

上面的代码说明，第一个数组的0号位置是有值的，第二个数组的0号位置是没有值的。

ECMA262V5对空位的处理，已经很不一致了，大多数情况下会忽略空位。比如，forEach()、filter()、every()和some()都会跳过空位；map()会跳过空位，但会保留这个值；join()和toString()会将空位视为undefined，而undefined和null会被处理成空字符串。

```
[,'a'].forEach((x,i) => log(i)); // 1
['a',,'b'].filter(x => true) // ['a','b']
[,'a'].every(x => x==='a') // true
[,'a'].some(x => x !== 'a') // false
[,'a'].map(x => 1) // [,1]
[,'a',undefined,null].join('#') // "#a##"
```

```
[,'a',undefined,null].toString() // ",a,,"
```

ECMA262V6则是明确将空位转为**undefined** 。比如，Array.from方法会将数组的空位转为undefined，也就是说，这个方法不会忽略空位。

```
Array.from(['a',,'b']) // [ "a", undefined, "b" ]
```

扩展运算符（...）也会将空位转为undefined。

```
[...['a',,'b']] // [ "a", undefined, "b" ]
```

copyWithin()会连空位一起拷贝。

```
[,'a','b',,].copyWithin(2,0) // [,"a",,"a"]
```

fill()会将空位视为正常的数组位置。

```
new Array(3).fill('a') // ["a","a","a"]
```

for...of循环也会遍历空位。

```
let arr = [, ,];
for (let i of arr) {   console.log(1); }
// 1
// 1
```

上面的代码中，数组arr有两个空位，for...of并没有忽略它们。如果改成map方法遍历，那么空位是会跳过的。

entries()、keys()、values()、find()和findIndex()会将空位处理成undefined。

```
[...[,'a'].entries()] // [[0,undefined], [1,"a"]]
[...[,'a'].keys()] // [0,1]
[...[,'a'].values()] // [undefined,"a"]
[,'a'].find(x => true) // undefined
[,'a'].findIndex(x => true) // 0
```

由于空位的处理规则非常不统一，所以建议避免出现空位 。

2.3　数值的扩展与修复

数值没有什么好扩展的，而且JavaScript的数值精度问题未修复，要修复它们可不是一两行代码了事。先看扩展，我们只把目光集中于Prototype.js与mootools就行了。

Prototype.js为它添加8个原型方法：Succ是加1；times是将回调重复执行指定次数toPaddingString与上面提到字符串扩展方法pad作用一样；toColorPart是转十六进制；abs、ceil、floor和abs是从Math中偷来的。

mootools的情况：limit是从数值限定在一个闭开间中，如果大于或小于其边界，则等于其最大值或最小值；times与Prototype.js的用法相似；round是Math.round的增强版，添加了精度控制；toFloat、toInt是从window中偷来的；其他的则是从

Math中偷来的。

在ES5_shim.js库中，它实现了ECMA262V5提到的一个内部方法toInteger。

```
// http://es5.github.com/#x9.4
// http://jsperf.com/to-integer
var toInteger = function(n) {
    n = +n;
    if (n !== n) { // isNaN
        n = 0;
    } else if (n !== 0 && n !== (1 / 0) && n !== -(1 / 0)) {
        n = (n > 0 || -1) * Math.floor(Math.abs(n));
    }
    return n;
};
```

但依我看来都没什么意义，数值往往来自用户输入，我们一个正则就能判定它是不是一个“数”。如果是，则直接Number（n）！

基于同样的理由，mass Framework对数字的扩展也是很少的，3个独立的扩展。

limit 方法：确保数值在［n1，n2］闭区间之

内，如果超出限界，则置换为离它最近的最大值或最小值。

```
function limit(target, n1, n2) {
    var a = [n1, n2].sort();
    if (target < a[0])
        target = a[0];
    if (target > a[1])
        target = a[1];
    return target;
}
```

nearer方法：求出距离指定数值最近的那个数。

```
function nearer(target, n1, n2) {
    var diff1 = Math.abs(target - n1),
            diff2 = Math.abs(target - n2);
    return diff1 < diff2 ? n1 : n2
}
```

Number下唯一需要修复的方法是toFixed，它是用于校正精确度，最后的数会做四舍五入操作，但在一些浏览器中并没有这样干。想简单修复的可以这样处理。

```
if (0.9.toFixed(0) !== '1') {
    Number.prototype.toFixed = function(n) {
        var power = Math.pow(10, n);
        var fixed = (Math.round(this * power) / power).toString
();
        if (n == 0)
            return fixed;
        if (fixed.indexOf('.') < 0)
            fixed += '.';
        var padding = n + 1 - (fixed.length - fixed.indexOf('.'
));
        for (var i = 0; i < padding; i++)
            fixed += '0';
        return fixed;
    };
}
```

追求完美的话，还存在这样一个版本，把里面的加、减、乘、除都重新实现了一遍。

https://github.com/es-shims/es5-shim/blob/master/es5-shim.js

toFixed方法实现得如此艰难其实也不能怪浏览器，计算机所理解的数字与我们是不一样的。众所周知，计算机的世界是二进制，数字也不例外。为了储存更复杂的结构，需要用到更高维的进制。而

进制间的换算是存在误差的。虽然计算机在一定程度上反映了现实世界，但它提供的顶多只是一个“幻影”，经常与我们的常识产生偏差。比如，将1除以3，然后再乘以3，最后得到的值竟然不是1；10个0.1相加也不等于1；交换相加的几个数的顺序，却得到了不同的和。JavaScript不能免俗。

```
console.log(0.1 + 0.2)
console.log(Math.pow(2, 53) === Math.pow(2, 53) + 1) //true
console.log(Infinity > 100) //true
console.log(JSON.stringify(25001509088465005)) //25001509088465
004
console.log(0.1000000000000000000000000001) //0.1
console.log(0.100000000000000000000000001) //0.1
console.log(0.1000000000000000000000000456) //0.1
console.log(0.09999999999999999999999) //0.1
console.log(1 / 3) //0.3333333333333333
console.log(23.53 + 5.88 + 17.64)// 47.05
console.log(23.53 + 17.64 + 5.88)// 47.050000000000004
```

这些其实不是bug，而是我们无法接受这事实。在JavaScript中，数值有3种保存方式。

（1）字符串形式的数值内容。

（2）IEEE 754标准双精度浮点数，它最多支持小数点后带15～17位小数，由于存在二进制和十进制的转换问题，具体的位数会发生变化。

（3）一种类似于C语言的int类型的32位整数，它由4个8 bit的字节构成，可以保存较小的整数。

当JavaScript遇到一个数值时，它会首先尝试按整数来处理该数值，如果行得通，则把数值保存为31 bit的整数；如果该数值不能视为整数，或超出31 bit的范围，则把数值保存为64位的IEEE 754浮点数。

聪明的读者一定想到了这样一个问题：什么时候规规矩矩的整数会突然变成捉摸不定的双精度浮点数？答案是：当它们的值变得非常庞大时，或者进入1和0之间时，规矩矩矩的整数就会变成捉摸不定的双精度浮点数。因此，我们需要注意以下数值。

首先是1和0；其次是最大的Unicode数值**1114111**（7位数字，相当于（/x41777777）；最大的RGB颜色值**16777215**（8位数字，相当于#FFFFFF）；最大的32 bit整数是**147483647**（10位数字，即`Math.pow(2,31)-1``）；最少的32位bit整数 **-2147483648**，因为JavaScript内部会以整数的形式保存所有Unicode值和RGB颜色；再次是**2147483647**，任何大于该值的数据将保存为双精度格式；最大的浮点数**9007199254740992**（16位数字，即Math.pow（2,53）），因为输出时类似整数，而所有Date对象（按毫秒计算）都小于该值，因此总是模拟整数的格式输出；最大的双精度数值**1.7976931348623157e+308**，超出这个范围就要算作无穷大了。

因此，我们就看出缘由了，大数相加出问题是由于精度的不足，小数相加出问题是进制转算时产生误差。第一个好理解，第二个，主要是我们常用

的十进制转换为二进制时，变成循环小数及无理数等有无限多位小数的数，计算机要用有限位数的浮点数来表示是无法实现的，只能从某一位进行截短。而且，因为内部表示是二进制，十进制看起来是能除尽的数，往往在二进制是循环小数。

比如用二进制来表示十进制的0.1，就得写成2的幂（因为小于1，所以幂是负数）相加的形式。若一直持续下去，0.1就成了0.00011001100110011001l…这种循环小数。在有效数字的范围内进行舍入，就会产生误差。

> 综上，我们就尽量避免小数操作与大数操作，或者转交后台去处理，实在避免不了就引入专业的库来处理。

2.4　函数的扩展与修复

ECMA262V5对函数唯一的扩展就是bind函数。众所周知，这是来自Prototype.js，此外，其他重要的函数都来自Prototype.js。

Prototype.js的函数扩展包括以下几种方法。

- argumentNames：取得函数的形参，以字符串数组形式返回。未来的Angular.js也是通过此方法实现函数编译与DI（依赖注入）。
- bind：劫持this，并预先添加更多参数。
- bindAsEventListener：如bind相似，但强制返回函数的第一个参数为事件对象，这是用于修复IE的多投事件API与标准API的差异。
- curry：函数柯里化，用于一个操作分成多步进行，并可以改变原函数的行为。
- wrap：AOP的实现。

- delay：setTimeout的“偷懒”写法。
- defer：强制延迟0.01s才执行原函数。
- methodize：将一个函数变成其调用对象的方法，这也是为其类工厂的方法链服务。

这些方法每一个都是别具匠心，影响深远。

我们先看bind方法，它用到了著名的闭包。所谓闭包，就是一个引用着外部变量的内部函数。比如下面这段代码。

```
var observable = function(val) {
    var cur = val;//一个内部变量
    function field(neo) {
        if (arguments.length) {//setter
            if (cur !== neo) {
                cur = neo;
            }
        } else {//getter
            return cur;
        }
    }
    field();
    return field;
}
```

上面代码里面的field函数将与外部的cur构成一

个闭包。Prototype.js中的bind方法只要依仗原函数与经过切片化的args构成闭包，而让这方法名符其实的是curry，用户最初的传参，劫持到返回函数修正this的指向。

```
Function.prototype.bind = function(context) {
    if (arguments.length < 2 && context == void 0)
        return this;
    var __method = this, args = [].slice.call(arguments, 1);
    return function() {
        return __method.apply(context, args.concat.apply(args,
arguments));
    }
}
```

正因为有这东西，我们才方便修复IE多投事件API和attachEvent回调中的this问题，它总是指向window对象，而标准浏览器的addEventListener中的this则为其调用对象。

```
var addEvent = document.addEventListener ?
        function(el, type, fn, capture) {
            el.addEventListener(type, fn, capture)
        } :
        function(el, type, fn) {
            el.attachEvent("on" + type, fn.bind(el, event))
        }
```

ECMA262V5对其认证后，唯一的增强是对调用者进行检测，确保它是一个函数。顺便总结一下。

（1）call是obj.method(a,b,c)到method(obj,a,b,c)的变换。

（2）apply是obj.method(a,b,c)到method(obj,[a,b,c])的变换，它要求第2个参数必须存在，一定是数组或Arguments这样的类数组，NodeList这样具有争议性的内容就不要乱传进去了。因此jQuery对两个数组或类数组的合并是使用jQuery.merge，放弃使用Array.prototype.push.apply。

（3）bind就是apply的变种，它可以劫持this对象，并且预先注入参数，返回后续执行方法。

这3个方法是非常有用，我们可以设法将它

们“偷”出来。

```
var bind = function(bind) {
    return{
        bind: bind.bind(bind),
        call: bind.bind(bind.call),
        apply: bind.bind(bind.apply)
    }
}(Function.prototype.bind)
```

那怎么用它们呢？比如我们想合并两个数组，直接调用concat，方法如下。

```
var a = [1, [2, 3], 4];
var b = [5,6];
console.log(b.concat(a)); //[5,6,1,[2,3],4]
```

使用**bind.bind** 方法则能将它们进一步平坦化。

```
var concat = bind.apply([].concat);
console.log(concat(b, a)); //[1,3,1,2,3,4]
```

又如切片化操作，它经常用于转换类数组对象

为纯数组的。

```
var slice = bind([].slice)
var array = slice({
    0: "aaa",
    1: "bbb",
    2: "ccc",
    length: 3
});
console.log(array)//[ "aaa", "bbb", "ccc"]
```

更常用的操作是转换arguments对象，目的是为了使用数组的一系列方法。

```
function test() {
    var args = slice(arguments)
    console.log(args)//[1,2,3,4,5]
}
test(1, 2, 3, 4, 5)
```

我们可以将hasOwnProperty提取出来，判定对象是否在本地就拥有某属性。

```
var hasOwn = bind.call(Object.prototype.hasOwnProperty);
hasOwn({a:1}, "a") // true
hasOwn({a:1}, "b") // false
```

使用**bind.bind** 就需要多执行一次。

```
var hasOwn2 = bind.bind(Object.prototype.hasOwnProperty);
hasOwn2({a:1}, "b")() // false
```

上面**bind.bind** 的行为其实就是一种**curry**，它给了你再一次传参的机会，这样你就可以在内部判定参数的个数，决定继续返回函数还是结果。这在设计计算器的连续运算上非常有用。从这个角度来看，我们可以得到一个信息，bind着重于作用域的劫持，curry在于参数的不断补充。

我们可以编写一个 curry，当所有步骤输入的参数个数等于最初定义的函数的形参个数时，就执行它。

```
function curry(fn) {
    function inner(len, arg) {
        if (len == 0)
            return fn.apply(null, arg);
        return function(x) {
            return inner(len - 1, arg.concat(x));
        };
    }
    return inner(fn.length, []);
```

```
}

function sum(x, y, z, w) {
    return x + y + z + w;
}
curry(sum)('a')('b')('c')('d'); // => 'abcd'
```

不过这里我们假定用户每次都只传入一个参数，所以我们可以改进一下。

```
function curry2(fn) {
    function inner(len, arg) {
        if (len <= 0)
            return fn.apply(null, arg);
        return function() {
            return inner(len - arguments.length,
                    arg.concat(Array.apply([], arguments)));
        };
    }
    return inner(fn.length, []);
}
```

这样就可以在中途传递多个参数，或不传递参数。

```
curry2(sum)('a')('b', 'c')('d'); // => 'abcd'
curry2(sum)('a')()('b', 'c')()('d'); // => 'abcd'
```

不过，上面的函数形式有个更帅气的名称，叫**self-curry** 或**recurry** 。它强调的是递归调用自身来补全参数。

与curry相似的是partial。curry的不足是参数总是通过push的方式来补全，而partial则是在定义时所有参数已经都有了，但某些位置上的参数只是个占位符，我们接下来的传参只是替换掉它们。博客上有篇文章《Partial Application in JavaScript》专门介绍了这个内容。

```
Function.prototype.partial = function() {
    var fn = this, args = Array.prototype.slice.call(arguments)
;
    return function() {
        var arg = 0;
        for (var i = 0; i < args.length && arg < arguments.leng
th; i++)
            if (args[i] === undefined)
                args[i] = arguments[arg++];
        return fn.apply(this, args);
    };
}
```

它是使用**undefined** 作为占位符。

```
var delay = setTimeout.partial(undefined, 10);
//接下来的工作就是代替掉第一个参数
delay(function() {
    alert("this call to will be temporarily delayed.");
})
```

有关这个占位符，该博客的评论列表中也有大量的讨论，最后确定下来是使用_作为变量名，内部还是指向undefined。笔者认为这样做还是比较危险的，框架应该提供一个特殊的对象，比如Prototype在内部使用$break = {}作为断点的标识。我们可以用一个纯空对象 作为partial的占位符。

```
var _ = Object.create(null)
```

纯空对象没有原型，没有toString、valueOf等继承自Object的方法，很特别。在IE下我们可以这样模拟它。

```
var _ = (function() {
    var doc = new ActiveXObject('htmlfile')
    doc.write('<script><\/script>')
    doc.close()
    var Obj = doc.parentWindow.Object
```

```
    if (!Obj || Obj === Object)
        return
    var name, names =
            ['constructor', 'hasOwnProperty', 'isPrototypeOf'
            , 'propertyIsEnumerable', 'toLocaleString', 'toStri
ng', 'valueOf']
    while (name = names.pop())
        delete Obj.prototype[name]
    return Obj
}())
```

我们继续回来讲partial。

```
function partial(fn) {
    var A = [].slice.call(arguments, 1);
    return A.length < 1 ? fn : function() {
        var a = Array.apply([], arguments);
        var c = A.concat();//复制一份
        for (var i = 0; i < c.length; i++) {
            if (c[i] === _) {//替换占位符
                c[i] = a.shift();
            }
        }
        return fn.apply(this, c.concat(a));
    }
}
function test(a, b, c, d) {
    return "a = " + a + " b = " + b + " c = " + c + " d = " + d
}
var fn = partail(test, 1, _, 2, _);
fn(44, 55)// "a = 1 b = 44 c = 2 d = 55"
```

curry、partial的应用场景在前端世界[2]真心不多，前端讲究的是即时显示，许多API都是同步

的，后端由于IO操作等耗时长，像Node.js提供了大量的异步函数来提高性能，防止堵塞。但是过多异步函数也必然带来回调嵌套的问题，因此我们需要通过curry等函数变换，将套嵌减少到可以接受的程度。这个我会在第13章讲述它们的使用方法。

函数的修复涉及apply与call两个方法。这两个方法的本质就是生成一个新的函数，将原函数与用户传参放到里面执行而已。在JavaScript创建一个函数有很多办法，常见的有函数声明和函数表达式，次之是函数构造器，再次是eval、setTimeout……

```
Function.prototype.apply || (Function.prototype.apply = functio
n (x, y) {
    x = x || window;
    y = y ||[];
    x.__apply = this;
    if (!x.__apply)
        x.constructor.prototype.__apply = this;
    var r, j = y.length;
    switch (j) {
        case 0: r = x.__apply(); break;
        case 1: r = x.__apply(y[0]); break;
        case 2: r = x.__apply(y[0], y[1]); break;
        case 3: r = x.__apply(y[0], y[1], y[2]); break;
        case 4: r = x.__apply(y[0], y[1], y[2], y[3]); break;
        default:
            var a = [];
```

```
            for (var i = 0; i < j; ++i)
                a[i] = "y[" + i + "]";
            r = eval("x.__apply(" + a.join(",") + ")");
            break;
    }
    try {
        delete x.__apply ? x.__apply : x.constructor.prototype.
__apply;
    }
    catch (e) {}
    return r;
});

Function.prototype.call || (Function.prototype.call = function
() {
    var a = arguments, x = a[0], y = [];
    for (var i = 1, j = a.length; i < j; ++i)
        y[i - 1] = a[i]
    return this.apply(x, y);
});
```

2.5　日期的扩展与修复

Date构造器是JavaScript中传参形式最丰富的构造器，大致分为4种。

```
new Date();
new Date(value);//传入毫秒数
new Date(dateString);
new Date(year, month, day /*, hour, minute, second, millisecond
*/);
```

其中第3种可以玩多种花样，个人建议只使用“2009/07/12 12:34:56”，后面的时分秒可省略。这个所有浏览器都支持。此构造器的兼容列表可见下文。

```
http://dygraphs.com/date-formats.html
```

若要修正它的传参，这恐怕是个大工程，要整个对象替换掉，并且影响Object.prototype.toString的

类型判定，因此不建议修正。ES5.js中有相关源码，大家可以看这里。

```
https://github.com/kriskowal/es5-shim/blob/master/es5-shim.js
```

JavaScript的日期是抄自Java的java.util.Date，但是Date这个类中的很多方法对时区等支持不够，且不少都是已过时的。Java程序员也推荐使用calnedar类代替Date类。JavaScript可选择的余地比较少，只能凑合继续用。比如：对属性使用了前后矛盾的偏移量，月份与小时都是基于0，月份中的天数则是基于1，而年则是从1900开始的。

接下来，我们为旧版本浏览器添加几个ECMA262标准化的日期方法吧。

```
if (!Date.now) {
    Date.now = function() {
        return +new Date;
    }
}
if (!Date.prototype.toISOString) {
  void function() {
```

```
     function pad(number) {
         var r = String(number);
         if (r.length === 1) {
             r = '0' + r;
         }
         return r;
    }

    Date.prototype.toJSON =
    Date.prototype.toISOString = function() {
       return this.getUTCFullYear()
               + '-' + pad(this.getUTCMonth() + 1)
               + '-' + pad(this.getUTCDate())
               + 'T' + pad(this.getUTCHours())
               + ':' + pad(this.getUTCMinutes())
               + ':' + pad(this.getUTCSeconds())
               + '.' + String((this.getUTCMilliseconds() / 1000
).toFixed(3)).slice(2, 5)
               + 'Z';
    };

  }();
}
```

IE6和IE7中，getYear与setYear方法都存在bug，不过这个修复起来比较简单。

```
if ((new Date).getYear() > 1900) {
    Date.prototype.getYear = function() {
        return this.getFullYear() - 1900;
    };
    Date.prototype.setYear = function(year) {
        return this.setFullYear(year); //+ 1900
    };
}
```

至于扩展，由于涉及本地化，许多日期库都需要改一改才能用，其中以dataFormat这个很有用的方法较为特别。笔者先给一些常用的扩展吧。

传入两个Date类型的日期，求出它们相隔多少天。

```
function getDatePeriod(start, finish) {
    return Math.abs(start * 1 - finish * 1) / 60 / 60 / 1000 /
24;
}
```

传入一个Date类型的日期，求出它所在月的第一天。

```
function getFirstDateInMonth(date) {
    return new Date(date.getFullYear(), date.getMonth(), 1);
}
```

传入一个Date类型的日期，求出它所在月的最后一天。

```
function getLastDateInMonth(date) {
```

```
    return new Date(date.getFullYear(), date.getMonth() + 1, 0)
;
}
```

传入一个Date类型的日期，求出它所在季度的第一天。

```
function getFirstDateInQuarter(date) {
    return new Date(date.getFullYear(), ~~(date.getMonth() / 3)
 * 3, 1);
}
```

传入一个Date类型的日期，求出它所在季度的最后一天。

```
function getFirstDateInQuarter(date) {
    return new Date(date.getFullYear(), ~~(date.getMonth() / 3)
 * 3 + 3, 0);
}
```

判断是否为闰年。

```
function isLeapYear(date) {
    return new Date(this.getFullYear(), 2, 0).getDate() == 29;
}
//EXT
function isLeapYear2(date) {
```

```
    var year = data.getFullYear();
   return !!((year & 3) == 0 && (year % 100 || (year % 400 == 0
 && year)));
}
```

取得当前月份的天数。

```
function getDaysInMonth1(date) {
    switch (date.getMonth()) {
        case 0:
        case 2:
        case 4:
        case 6:
        case 7:
        case 9:
        case 11:
            return 31;
        case 1:
            var y = date.getFullYear();
            return y % 4 == 0 && y % 100 != 0 || y % 400 == 0 ?
 29 : 28;
        default:
            return 30;
    }
}

var getDaysInMonth2 = (function() {
    var daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31];

    function isLeapYear(date) {
        var y = date.getFullYear();
        return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
    }
    return function(date) { // return a closure for efficiency
        var m = date.getMonth();

        return m == 1 && isLeapYear(date) ? 29 : daysInMonth[m]
;
    };
```

```
})();

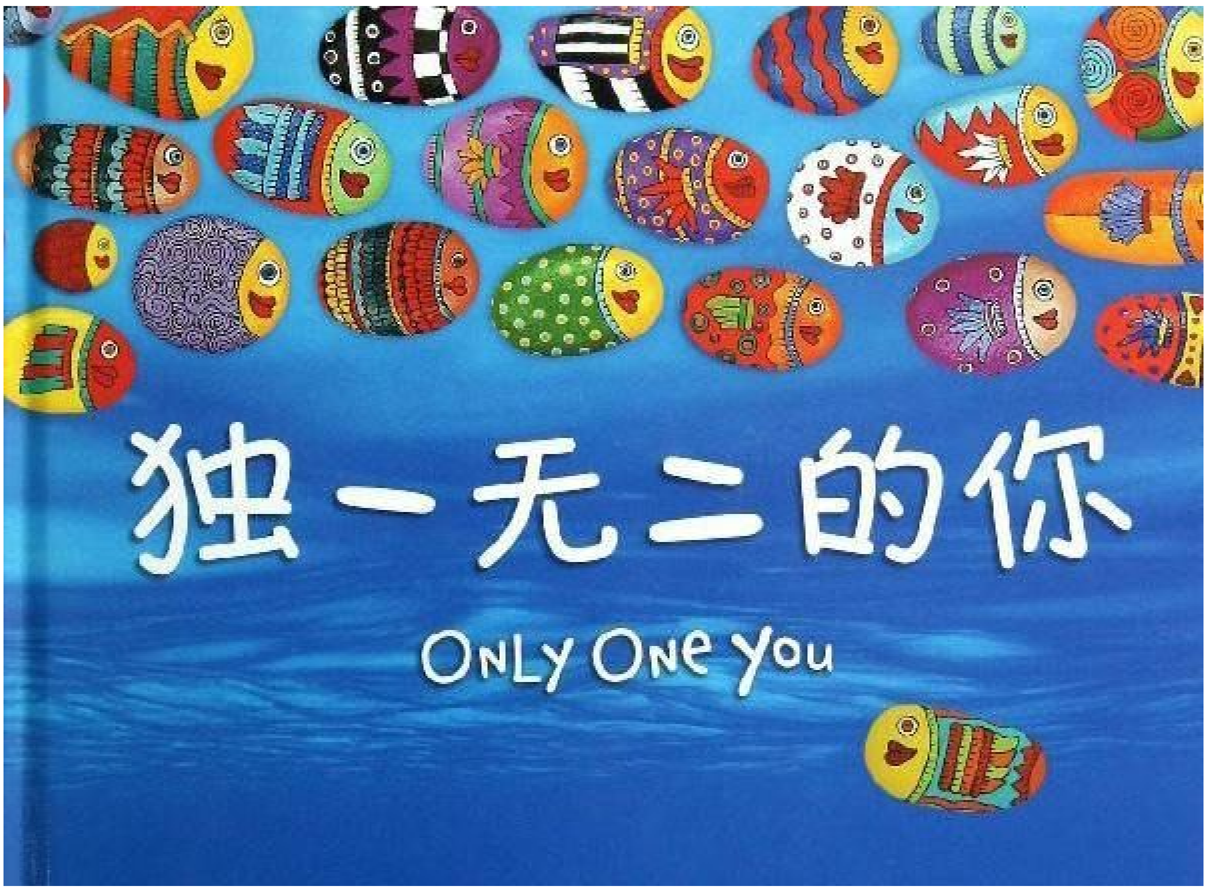
function getDaysInMonth3(date) {
    return new Date(date.getFullYear(), date.getMonth() + 1, 0)
.getDate();
}
```

---

[1] http://imququ.com/post/bom-and-javascript-trim.html

[2] 在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数且返回结果的新函数的技术。这个技术由Christopher Strachey以逻辑学家Haskell Curry命名的，尽管它是Moses Schnfinkel和Gottlob Frege发明的。patial，bind只是其一种变体。其用处有3：1.参数复用；2.提前返回；3.延迟计算/运行。

第3章　浏览器嗅探与特征侦测



本章完全是服务节点操作的模块，是处理兼容性的前线阵地。人们学会navigator. userAgent来判定用户正在使用的浏览器，然后选择性地去支持A

浏览器或放弃B浏览器。这导致浏览器厂商非常紧张，设法伪装自己为某一个主流浏览器。2010～2014年，中国互联网爆发著名的“3Q大战”，其间360安全浏览器无法访问QQ空间，作为对策，360将所有与自己相关的特征都抹去，伪造成一个普通的chrome浏览器，这标志着浏览器嗅探技术彻底失败。

时至今日，浏览器嗅探已经不推荐了，但在某些场合我们还是需要的，比如一些数据统计代码，著名的有站长统计、百度统计、腾讯统计和Google Analytics。而特征侦测是Prototype时期发展起来的一个技术，具体是判定某个原生对象有没有此方法或属性，有时严格一些，则会执行这个方法，看它有没有达到预期的值。在标准浏览器里，它们提供了document.implementation.hasfeature方法，可惜有bug，不准确。在本书快成形时，W3C又推出了CSS.supports方法，为大家又提供一条处理兼容性

问题的新道路，如图3-1所示。

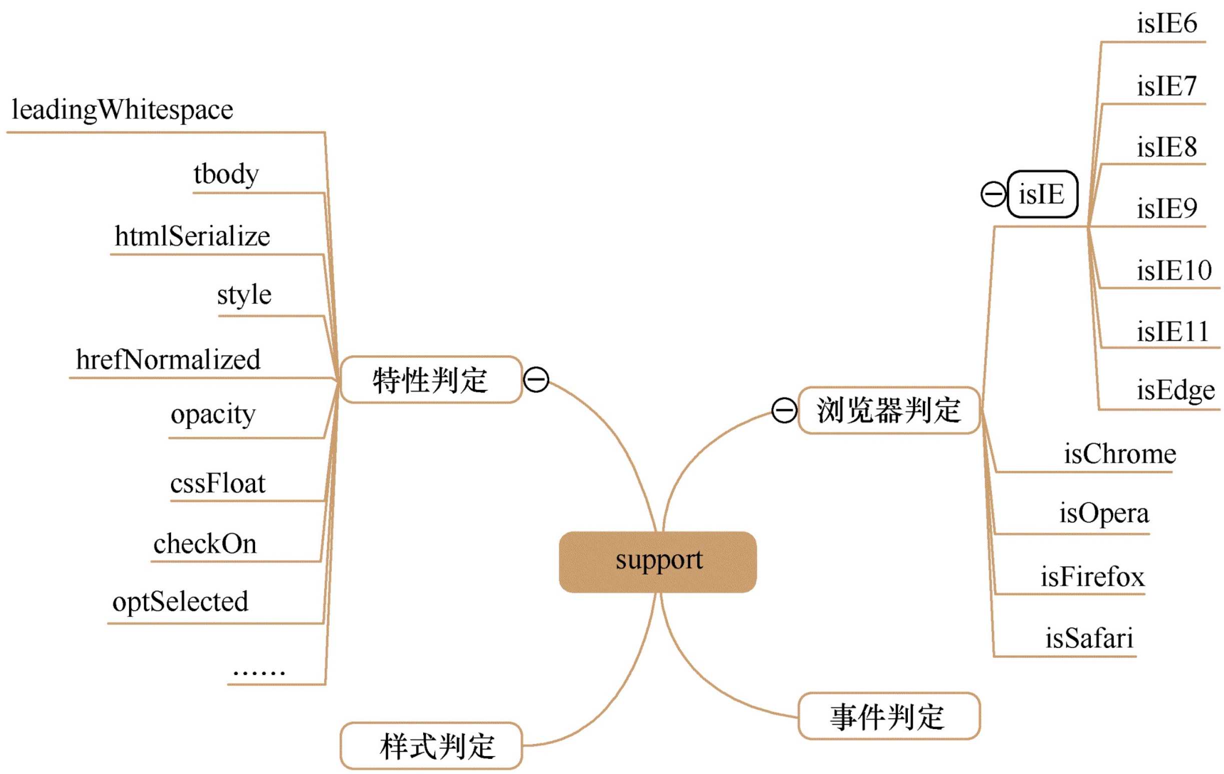

support
特性判定
leadingWhitespace
tbody
htmlSerialize
style
hrefNormalized
opacity
cssFloat
checkOn
optSelected
......
浏览器判定
isIE
isIE6
isIE7
isIE8
isIE9
isIE10
isIE11
isEdge
isChrome
isOpera
isFirefox
isSafari
事件判定
样式判定

图3-1

3.1 浏览器判定

主流浏览器有 5 个：**IE** 、**Firefox** 、**Opera** 、**Chrome** 和**Safari** 。早期所有框架都是通过 `navigator.userAgent` 进行判定，与浏览器厂商斗智斗谋。

jQuery给出的解决方案`jQuery.browser`，目前已移出框架本体，成为一个插件。

```
(function(jQuery, window, undefined) {
    "use strict";
    var matched, browser;

    jQuery.uaMatch = function(ua) {
        ua = ua.toLowerCase();

        var match = /(chrome)[ \/]([\w.]+)/.exec(ua) ||
                /(webkit)[ \/]([\w.]+)/.exec(ua) ||
                /(opera)(?:.*version|)[ \/]([\w.]+)/.exec(ua) |
|
                /(msie) ([\w.]+)/.exec(ua) ||
                ua.indexOf("compatible") < 0 && /(mozilla)(?:.*
? rv:([\w.]+)|)/.exec(ua) ||
                [];

        var platform_match = /(ipad)/.exec(ua) ||
                /(iphone)/.exec(ua) ||
                /(android)/.exec(ua) ||
                [];
```

```
        return {
           browser: match[ 1 ] || "",
           version: match[ 2 ] || "0",
           platform: platform_match[0] || ""
        };
    };

    matched = jQuery.uaMatch(window.navigator.userAgent);
    browser = {};

    if (matched.browser) {
        browser[ matched.browser ] = true;
        browser.version = matched.version;
    }

    if (matched.platform) {
        browser[ matched.platform ] = true
    }

// Chrome is Webkit, but Webkit is also Safari.
    if (browser.chrome) {
        browser.webkit = true;
    } else if (browser.webkit) {
        browser.safari = true;
    }

    jQuery.browser = browser;

})(jQuery, window);
```

mass Framework给出的解决方案如下。

```
//https://github.com/RubyLouvre/mass-Framework/blob/1.4/more/br
ower.js
define("brower", function( ){
     var w = window,ver = w.opera ? (opera.version().replace(/\
d$/, "") - 0)
    : parseFloat((/(?:IE |fox\/|ome\/|ion\/)(\d+\.\d)/.
        exec(navigator.userAgent) || [,0])[1]);
```

```
    return {
        //测试是否为IE或内核为trident，是则取得其版本号
        ie: !!w.VBArray && Math.max(document.documentMode||0, v
er),//内核trident
        //测试是否为Firefox，是则取得其版本号
        firefox: !!w.netscape && ver,//内核Gecko
        //测试是否为Opera，是则取得其版本号
        opera:  !!w.opera && ver,//内核 Presto 9.5为Kestrel 10为C
arakan
        //测试是否为Chrome，是则取得其版本号
        chrome: !! w.chrome &&  ver ,//内核V8
        //测试是否为Safari，是则取得其版本号
        safari: /apple/i.test(navigator.vendor) && ver// 内核 We
bCore
    }
});
```

mass成形较晚，是使用特征侦测实现的。

特征侦测的好处是，浏览器不会随意去掉某一个功能。不过要注意的是，不要使用标准属性与方法做判定依据。每个浏览器都有自己的私有实现，我们用它们做判定就可以了。下面是我收集的其他一些判定方法。

```
ie = !!document.recalc
ie = !!window.VBArray
ie = !!window.ActiveXObject
ie = !!window.createPopup;
ie = /*@cc_on!@*/!1;
ie = document.expando;//document.all在opera firefox的古老版本也存
在
```

```
ie = (function() {//IE10中失效
    var v = 3, div = document.createElement('div');
    while (div.innerHTML = '<!--[if gt IE ' + (++v) + ']><br><!
[endif]-->', div.innerHTML )
        ;
    return v > 4 ? v : !v;
}());

ie678 = !+"\v1";
ie678 = !-[1, ];
ie678 = '\v' == 'v';
ie678 = ('a~b'.split(/(~)/))[1] == "b"
ie678 = 0.9.toFixed(0) == "0"
ie678 = /\w/.test('\u0130') //由群里的abcd友情提供
ie8 = window.toStaticHTML
ie9 = window.msPerformance

ie678 = 0//@cc_on+1;

ie67 = !"1"[0] //利用IE6或IE5的字符串不能使用数组下标的特征
ie8 = !!window.XDomainRequest;
ie9 = document.documentMode && document.documentMode === 9;

//基于条件编译的嗅探脚本，IE会返回其JS引擎的版本号，非IE返回0
var ieVersion = eval("''+/*@cc_on" + " @_jscript_version@*/-0")
 * 1
ie9 = ieVersion === 5.9
ie8 = ieVersion === 5.8
ie7 = ieVersion === 5.7
ie6 = ieVersion === 5.6
ie5 = ieVersion === 5.5
ie10 = window.navigator.msPointerEnabled
ie11 = '-ms-scroll-limit' in document.documentElement.style
opera = !!window.opera;
//https://developer.mozilla.org/En/Windows_Media_in_Netscape
firefox = !!window.GeckoActiveXObject
firefox = !!window.netscape //包括firefox
firefox = !!window.Components
firefox = !!window.updateCommands
safari = !!(navigator.vendor && navigator.vendor.match(/Apple/)
)
safari = window.openDatabase && !window.chrome;
chrome = !!(window.chrome && window.google)
```

至于移动设备的相关判定，建议看百度fex-team出品的ua-device库。

```
https://github.com/fex-team/ua-device
isIPhone = /iPhone/i.test(navigator.userAgent);
isIPhone4 = window.devicePixelRatio >= 2
isIPad = /iPad/i.test(navigator.userAgent);
isAndroid = /android/i.test(navigator.userAgent);
isIOS = isIPhone || isIPad ;
```

这里解释一下iPhone的判定。在网页中，pixel与point比值称为device-pixel-ratio，普通设备都是1，iPhone 4是2，有些Android机型是1.5。

3.2　document.all趣闻

上面笔者给出了一系列特征嗅探的方法，可能会让一些人不解，为什么没有document.all呢？document.all是IE下一个很好用的选择器API。大概是1997年，IE4发布时带来的最好用的API之一。既然这么好用，肯定会被人抄去。但很长时间内，它还是IE独有的。一些旧的技术书与博客，介绍它为IE的判定手段也是没有错，但到2002年，出现了转机。

Opera这个小众的浏览器，为了生存，总是兼容两套API：IE的与w3c的。比如说IE的attachEvent，它也是最早支持。当Opera实现了document.all后，这个通用的IE判定就失灵了，许多开发者给Opera提issue，人家并不想修复。

此后，Firefox也决定实现document.all，不过它

耍了一个小聪明，那就是你可以正常的使用document.all，但你无法检测到它的存在。

```
console.log(document.all + "" )
//"[object HTMLAllCollection]"
console.log(typeof document.all)
//"undefined"
console.log(!!document.all)
//false
```

JavaScript之父Brendan Eich称其为“undetected document.all”。但在当时很多人也发现了，document.all并不是真的检测不到。

```
console.log(document.all === undefined)
//false
console.log("all" in document)
//true
```

当时Mozilla的人也回复了：“这不是bug，这是feature!!!”。现如今所有的浏览器都是这样的实现，HTML 5规范里也是这么规定的。

```
https://www.w3.org/TR/html5/obsolete.html#dom-document-all
```

Safari才刚刚起步，但也有收到来自用户关于不支持document.all的bug。2005年年底，Safari学Firefox，实现了undetectable document.all。

```
https://bugs.webkit.org/show_bug.cgi?id=6268
```

2008年，Opera在9.50 Beta 2版本将自己直接暴露了多年的document.all也改成了undetectable，变更记录里是这么写的："Opera now cloaks document.all"。Opera的工程师当年还专门写了一篇文章讲了document.all在Opera中的变迁，还说到document.all绝对值得被展览进"Web技术博物馆"。

2008年年底，Chrome 1.0发布，Chrome是基于Webkit和V8的，V8当然得配合Webkit里的document.all的实现。

很戏剧性的是，在2013年，连IE自己（IE 11）

也隐藏掉了document.all，也就是说，所有现代浏览器里document.all都是个假值了。

在V8里的实现是：一个对象都可以被标记成undetectable，很多年来只有document.all带有这个标记，这是相关的代码片段和注释。

```
// Tells whether the instance is undetectable.
// An undetectable object is a special class of JSObject: 'type
of' operator
// returns undefined, ToBoolean returns false. Otherwise it beh
aves like
// a normal JS object.  It is useful for implementing undetecta
ble
// document.all in Firefox & Safari.
// See https://bugzilla.mozilla.org/show_bug.cgi?id=248549.
inline void set_is_undetectable();
inline bool is_undetectable();
```

然后在typeof的实现里，如果typeof的参数是undefined或者是undetectable，就返回“undefined”。

当然，我们在JS中很难判定哪些是undetectable，但undetectable的内容都是与DOM和BOM有关，那里是浏览器可以自由发挥的领域。并

且从document.all的变更史看来，基本特征嗅探的代码，有点让人心惊胆战。这些方法也足以验证一个人在这个领域的耕耘之深。

3.3 事件的支持侦测

Prototype的核心成员kangax写了一篇叫《Detecting event support without browser sniffing》文章，来判定浏览器对某种事件的支持，里面给出的实现如下。

```
var isEventSupported = (function() {
    var TAGNAMES = {
        'select': 'input', 'change': 'input',
        'submit': 'form', 'reset': 'form',
        'error': 'img', 'load': 'img', 'abort': 'img'
    }
    function isEventSupported(eventName) {
        var el = document.createElement(TAGNAMES[eventName] ||
'div');
        eventName = 'on' + eventName;
        var isSupported = (eventName in el);
        if (!isSupported) {
            el.setAttribute(eventName, 'return;');
            isSupported = typeof el[eventName] == 'function';
        }
        el = null;
        return isSupported;
    }
    return isEventSupported;
})();
```

现在jQuery与mass使用的脚本都是其简化版，

其中mass对IE内存泄漏做了优化。

```
$.eventSupport = function(eventName, el) {
    el = el || document.documentElement
    eventName = "on" + eventName;
    var ret = eventName in el;
    if (el.setAttribute && !ret) {
        el.setAttribute(eventName, "");
        ret = typeof el[eventName] === "function";
        el.removeAttribute(eventName);
    }
    el = null;
    return ret;
}
```

不过哪一个也好，这种检测只对DOM0事件凑效，像DOMMouseScroll、DOMContentLoaded、DOMFocusIn、DOMFocusOut、DOMSubtreeModified、DOMNodeInserted、DOMNodeRemoved这些以DOM开头的事件就无能为力了。

注意：上面提到的许多以DOM开头的事件都是已被废弃的DOM4变动事件**MutationEvents**，只有chrome还继续支持它们。因此大家不用费心记住

它们。

```
* DOMAttrModified
* DOMAttributeNameChanged
* DOMCharacterDataModified
* DOMElementNameChanged
* DOMNodeInserted
* DOMNodeInsertedIntoDocument
* DOMNodeRemoved
* DOMNodeRemovedFromDocument
* DOMSubtreeModified
```

这些事件中有的非常有用，比如：DOMMouseScroll，Firefox一直不支持mousewheel，只能用它做替代品；DOMContentLoaded是实现domReady的重要事件；DOMNodeRemoved是判定元素是否从其父节点移除，父节点可能是其他元素节点或文档碎片；DOMNodeRemovedFromDocument是移离DOM树；DOMAttrModified以前经常用于模拟IE的onpropertychange；DOMCharacterDataModified用于监听contenteditable为true的元素内容变动。

在mass Framework中，是使用以下方法判定的。

```
https://github.com/RubyLouvre/mass-Framework/blob/1.4/event.js
try {
    //如果浏览器支持创建MouseScrollEvents事件对象，那么就用DOMMouseScroll
    document.createEvent("MouseScrollEvents");
    eventHooks.mousewheel = {
        bindType: "DOMMouseScroll",
        delegateType: "DOMMouseScroll"
    };
    //如果某一天,Firefox回心转意支持mousewheel，那么我们就不需要这个钩子
    if ($.eventSupport("mousewheel")) {
        delete eventHooks.mousewheel;
    }
} catch (e) {
}
```

此外，mass还对focusin进行识别。focusin与focusout是一对，判定当中一个就明白另一个情况。这两个事件也很重要，用于实现focus与blur的事件代理，因为focus与blur不支持冒泡，需要用它们的冒泡版实现（假若不支持focusin与focusout，jQuery也找到办法了，不过有原生的就用原生的）。

```
//首先判定它是否是W3C阵营，IE肯定支持
$.support.focusin = !!window.attachEvent;
$(function() {
    var div = document.createElement("div");
    document.body.appendChild(div);
    div.innerHTML = "<a href='#'></a>";
    if (!$.support.focusin) {
        a = div.firstChild;
        a.addEventListener('focusin', function() {
           $.support.focusin = true;
        }, false);
        a.focus();
    }
});
```

CSS3添加两种动画，一种是**transition** 渐变动画，一种是**animation** 补间动画，它们在结束时都有相应的事件回调。但在标准化过程中，浏览器给它们起的名字相当没规律。这个也需预先侦测出来。

下面是**bootstrap** 的实现，听说来源于**modernizr**，比较粗糙。比如说你现在用的Opera已经支持不带前缀的标准事件名，它还是返回oTransitionEnd。

```
$.support.transition = (function() {
```

```
    var transitionEnd = (function() {
        var el = document.createElement('bootstrap'),
                transEndEventNames = {
            'WebkitTransition': 'webkitTransitionEnd',
            'MozTransition': 'transitionend',
            'OTransition': 'oTransitionEnd otransitionend',
            'transition': 'transitionend'
        };
        for (var name in transEndEventNames) {
            if (el.style[name] !== undefined) {
                return transEndEventNames[name]
            }
        }
    }())
    return transitionEnd && {
        end: transitionEnd
    }
})()
```

animation 补间动画 的检测可以看一下avalon的effect模块。

```
//animationend有两个可用形态
//IE10+, Firefox 16+ & Opera 12.1+: animationend
//Chrome/Safari: webkitAnimationEnd
//http://blogs.msdn.com/b/davrous/archive/2011/12/06/introducti
on-to-css3-animat ions.aspx
//IE10也可以使用MSAnimationEnd监听，但是回调里的事件 type依然为animati
onend
//  el.addEventListener("MSAnimationEnd", function(e) {
//     alert(e.type)// animationend！！！
// })
var checker = {
    'AnimationEvent': 'animationend',
    'WebKitAnimationEvent': 'webkitAnimationEnd'
}
var ani
for (name in checker) {
```

```
    if (window[name]) {
        ani = checker[name];
        break;
    }
}
if (typeof ani === "string") {
    supportAnimation = true
    animationEndEvent = ani
}
```

3.4 样式的支持侦测

CSS3带来许多好用的样式，但麻烦的是每个浏览器都有自己的私有前缀（或叫厂商前缀）。mass Framework与avalon提供了一个cssName方法处理它们，如果存在，则返回可用的驼峰风格样式名；如果不存在，就返回null。

```
var prefixes = ['', '-webkit-', '-o-', '-moz-', '-ms-'];
var cssMap = {
    "float": $.support.cssFloat ? 'cssFloat' : 'styleFloat',
    background: "backgroundColor"
};
function cssName(name, host, camelCase) {
    if (cssMap[name]) {
        return cssMap[name];
    }
    host = host || document.documentElement
    for (var i = 0, n = prefixes.length; i < n; i++) {
        camelCase = $.String.camelize(prefixes[i] + name);
        if (camelCase in host) {
            return (cssMap[name] = camelCase);
        }
    }
    return null;
}
```

一个样式名可以对应多种样式值，比如display

的值就有多种取值。如果要探知浏览器是否支持某一种，可以通过一个叫CSS.supports的API查询。如果不支持，则尝试一下这个开源项目，显然它还有很多探不出来的。

```
https://github.com/termi/CSS.supports
```

3.5　jQuery一些常用特征的含义

jQuery在support模块列举了一些常用的DOM特征的支持情况，不过它们的名字起得很奇怪，这里逐一揭开它们的谜底。由于这些特征在jQuery不同版本变动很大，本书以jQuery1.8为准。

- **leadingWhitespace**：判定浏览器在进行innerHTML赋值时，是否存在trimLeft操作。这个原本是IE出来搞的，应该遵循IE的游戏规则才对。结果其他浏览器一直认为要忠于用户的原始值，最前的空白不能省掉，要变成一个文本节点。最后微软将IE6、IE7、IE8返回false，其他浏览器返回true。
- **tbody**：指在用innerHTML动态创建元素时，浏览器会在多个tr元素外面包裹一个tbody元素（如果你没有添加的话），但如果这只是一个空table元素，标准浏览器是不会插入tbody的，

而旧式IE则没加判定地乱添加tbody元素，导致我们的业务代码与预期的不一样。自动插入tbody这一特征是非常棒的，在表格布局的年代，如果没有tbody，table会在浏览器解析到闭合标签时才显示出来。换言之，如果这个表格很大、很长，用户会什么也看不到。但有了tbody分段识别和显示，避免了页面长时间一片空白后突然一下子内容全部出来的局面。看下面的实验。

```
var div = document.createElement("div");
div.innerHTML = "<table></table>"
alert(div.innerHTML)
//IE678返回 "<TABLE><TBODY></TBODY></TABLE>"
//其他返回"<table></table>"
```

- **htmlSerialize**：判定浏览器是否完好支持用innerHTML转换一个符合html标签规则的字符串为一个元素节点，此过程jQuery称之为序列化。但IE支持不完好，包括script、link、style、meta

在内的no-scope元素都转换失败，需要在它前面添加一些字符，如"x<script src="xxx"><\/script>"，"<script src="xxx"><\/script>"或者"<br class= 'remove'><script src="xxx"><\/script>"。像HTML5的新标签，当然也支持不好，侦测结果同上。

- **style**：这个命名很难懂，不看源码不知道是什么意思。真相是判定getAttribute能否返回style的用户预设值。IE6、IE7、IE8没有区分特性属性，返回一个CSSStyleDeclaration对象。
- **hrefNormalized**：同样莫名其妙，意为判定getAttribute能否返回href的用户预设值。IE会多此一举，补充为完整路径给你。
- **opacity**：判定是否支持opacity样式值。IE6、IE7、IE8不支持，要用透明滤镜。
- **cssFloat**：判定float样式值在DOM的名字是哪个，W3C为cssFloat，IE6、IE7、IE8为

styleFloat。

- **checkOn：** 在大多数浏览器中checkbox的value默认为on，唯有Chrome返回空字符串。
- **optSelected：** 判定能否正确地取得动态添加的option元素的selected。IE6～IE10与老版本的Safari对动态添加的option没有设置为true。解决办法为，在访问selected属性前，先访问其父节点的selectedIndex属性，强制它计算option的selected。

```
<select id="optSelected">
</select>
<script type="text/javascript">
    var select = document.getElementById('optSelected');
    var option = document.createElement('option');
    select.appendChild(option);
    alert(option.selected);
    select.selectedIndex;
    alert(option.selected);
</script>
```

- **optDisabled：** 判定select元素的disabled属性是否影响到子元素的disabled取值。在Safari中，一

且select元素被disabled，它的孩子也被disabled，导致一个值也取不到。

- **checkClone：** 是指一个checkbox元素，如果设置了checked=true，且在多次克隆后，它的复制品能否保持为true。这个只有在Safari4中返回false，其他为true。

inlineBlockNeedsLayout： 判定是否使用hasLayout方法让display:inline-block生效。这个只有IE6、IE7、IE8为true。

- **getSetAttribute：** 判定是否区分特性属性（atttribute与property）。只有IE6、IE7、IE8为false。
- **noCloneEvent：** 判定在复制元素时是否复制attachEvent绑定的事件，只有旧版本IE及其兼容模式返回false。
- **enctype：** 判定浏览器是否支持encoding属性，

IE6、IE7要用encoding属性代替。

- **boxModel**：判定浏览器是否在content-box盒子模型的渲染模式下。submitBubbles，changeBubbles，focusinBubbles：判定浏览器是否支持这些事件，一直冒泡到document。
- **shrinkWrapBlocks**：判定元素是否会被子元素撑开。在IE6、IE7、IE8中，非替换元素[1]在设置了大小与hasLayout的情况下，会将其父级元素撑大。
- **html5Clone**：判定能否使用cloneNode复制HTML5的新标签，旧版本IE不支持，需要用到outerHTML。
- **deleteExpando**：判定能否删除元素节点上的自定义属性[2]，这用于jQuery缓存系统。旧版本IE不支持，会抛错，只能置换为undefined。
- **pixelPosition**：判定getComputedStyle能否转换元素的top、left、right、bottom的百分比值。这个webkit系出现问题，需要用到Dean Edwards大

神的hack。

- **reliableMarginRight**：判定getComputedStyle能否正确取得元素的marginRight。Safari的早期版本总是取回一个很大的数。
- **clearCloneStyle**：IE9、IE10出现的奇葩bug，当复制一个指定了background-*样式的元素，对复制品的背景进行清空时，也会清空原来的。

目前就是这么多了，随着浏览器疯狂更新版本，标准浏览器引发的各种bug已超越IE了。特征侦测不退反进，越来越重要了。

---

[1] 替换元素是浏览器根据其标签的元素与属性来判断显示具体的内容。img、input、textarea、select、object等都是替换元素，这些元素都没有实际的内容。替换元素可以增加行框高度，但不增加line-height。剩下的是不可替换元素，他们将内容直接告诉浏览器，将其显示出来。比如p的内容、

label的内容，浏览器将把这段内容直接显示出来。非替换元素添加padding-top或padding-bottom，不影响行框高度，但内容区高度会变化，margin-top，margin-bottom对行框没有任何影响。添加左右边距会影响非替换元素水平位置。要使非替换元素在父元素框内居中，可以设定line-height =父元素框的高度。

[2]　要理解delete操作符在浏览器上是如何运算是一个非常复杂的问题。当然我声明一个变量时，它实际上成为全局变量的某个属性。既然成了属性，就拥有属性的某些内部特质——ReadOnly, DontEnum, DontDelete和Internal。某个属性不能删除（删除不成功或干脆抛错），就是由DontDelete这个特性决定的。内建对象的一些属性拥有内部属性DontDelete，因此不能被删除；特殊的arguments变量（如我们所知的，活化对象的属性）拥有DontDelete；任何函数实例的length（返回形参长

度）属性也拥有DontDelete；与函数arguments相关联的属性也拥有DontDelete，同样不能被删除。

第4章　类工厂



类与继承在JavaScript的出现，说明JavaScript已经到达大规模开发的门槛了。在此之前的ES4，就试图引入类、模块等东西，也只不过把类延迟到ES6。到目前为止，只有少量浏览器支持它，当然

我们也可以直接上 ES6，然后使用babel编译我们的源码。曾经一段时间，类工厂是框架的标配。本章将会介绍如何模拟类及如何使用ES6的真正类。

4.1　JavaScript对类的支撑

在其他语言中，类的实例都要通过构造函数new出来。作为一个刻意模仿Java的语言，JavaScript存在new操作符，并且它的所有函数都可以作为构造器。构造器与普通的方法没有什么区别。浏览器为了构建它繁花似锦的生态圈，比如Node、Element、HTMLElement、HTMLParagraphElement，显然使用继承关系方便一些方法或属性的共享，于是JavaScript从其他语言借鉴了原型这种机制。

prototype 作为一个特殊的对象属性存在于每一个函数上。当一个函数通过new操作符“分娩”出其孩子——“实例”，这个名为实例的对象就拥有这个函数的prototype对象所有的一切成员，从而实现所有实例对象都共享一组方法或属性。而JavaScript所谓的“类”就是通过修改这个prototype对象，以区

别原生对象及其他自定义“类”。在浏览器中，Node这个类就是基于Object修改而来的，Element则是基于Node，而HTMLElement又基于Element……相对于我们的工作业务，我们也可以创建自己的类来实现重用与共享，如图4-1所示。

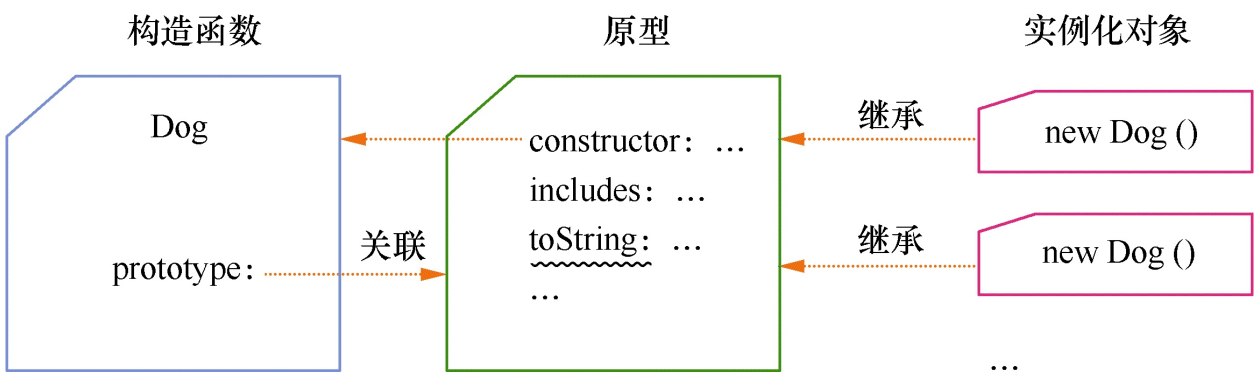


图4-1

```
function Dog() {}
Dog.prototype = {
    aa: "aa",
    method: function() {
    }
};
var a = new Dog;
var b = new Dog;
console.log(a.aa === b.aa);//true
console.log(a.method === b.method);//true
```

一般地，我们把定义在原型上的方法叫原型方

法，它为所有实例所共享。这有好处也有坏处，为了实现差异化，JavaScript允许我们直接在构造器内指定其方法，这叫做特权方法 。如果是属性，就叫特权属性。它们每一个实例就是一个副本，各不影响。因此我们通常把共享的用于操作数据的方法放在原型，把私有的属性放在特权属性中。但放于this上，还是能让人任意访问到，那就放在函数体内的作用域内吧。这时它就成为名副其实的私有属性 。

```
function A() {
    var count = 0
    this.aa = "aa";
    this.method = function() {
      return count
    }
    this.obj = {}
}

A.prototype = {
    aa: "aa",
    method: function() {
    }
};
var a = new A;
var b = new A;
console.log(a.aa === b.aa);  //true 由于aa的值为基本类型,比较值
console.log(a.obj === b.obj);//false,引用类型,每次进入函数体都重新创建，因此都不一样
console.log(a.method === b.method);//false
```

特权方法或属性只是遮住原型方法或属性，因此只要删掉特权方法，就又能访问到同名的原型方法或属性。

当访问一个对象的属性时，JavaScript会从对象本身开始往上遍历整个原型链，直到找到对应属性为止。如果此时到达了原型链的顶部，也就是Object.prototype，仍然未发现需要查找的属性，那么JavaScript就会返回undefined值。

```
delete a.method;
console.log(a.method === A.prototype.method);//true
```

用Java的语言来说，原型方法与特权方法都属于实例方法，在Java中还有一种叫做类方法 与类属性 的东西。它们用JavaScript来模拟也非常简单，直接定义在函数上就行了。

```
A.method2 = function(){};//类方法
var c = new A;
console.log(c.method2);//undefined
```

接下来，我们看一下继承 的实现。上面说过，只要prototype有什么东西，它的实例就有什么东西，不论这个属性是自带的，还是后来添加的。甚至，我们将这个prototype对象置换为另一个类的原型，这个对象都能轻而易举得到那个类的所有原型成员。

```
function A(){}
A.prototype = {
    aaa:1
}
function B(){}
B.prototype = A.prototype;
var b= new B;
console.log(b.aaa);//1;
A.prototype.bbb = 2;
console.log(b.bbb);//2;
```

由于是引用相同的一个对象，这意味着，如果我们修改A类的原型，也等同于修改了B类的原型。因此我们不能把一个对象赋给两个类。这有两

种办法：方法一，通过for in把父类的原型成员逐一赋给子类的原型；方法二，子类的原型不是直接由父类获得，先将此父类的原型赋给一个函数，然后将这个函数的实例作为子类的原型。

方法一，我们通常要实现**mixin** 这样的方法，亦有书称之为拷贝继承，好处是简单直接，坏处是无法通过instanceof验证。Prototype.js的extend方法就用来干这事。

```
function extend(destination, source) {
  for(var property in source)
  destination[property] = source[property];
  return destination;
}
```

方法二，就在原型上动脑筋，因此称之为原型继承。下面是个范本。

```
A.prototype = {
  aa: function() {
    alert(1)
  }
}
```

```
function bridge() {};
bridge.prototype = A.prototype;

function B() {}
B.prototype = new bridge();
var a = new A;
var b = new B;
//false，说明成功分开它们的原型
console.log(A.prototype == B.prototype);
//true，子类共享父类的原型方法
console.log(a.aa === b.aa);
//为父类动态添加新的原型方法
A.prototype.bb = function() {
  alert(2)
}
//true，孩子总会得到父亲的遗产
console.log(a.bb === b.bb);
B.prototype.cc = function() {
  alert(3)
}
//false，但父亲未必有机会看到孩子的新产业
console.log(a.cc === b.cc);
//并且它能正常通过JavaScript自带验证机制——instanceof
console.log(b instanceof A);//true
console.log(b instanceof B);//true
```

并且，方法二能通过instanceof验证。现在ES5就内置了这种方法来实现原型继承，它就是Object.create。如果不考虑第二个参数，它约等于下面的代码。

```
Object.create = function (o) {
    function F() {}
    F.prototype = o;
    return new F();
```

```
}
```

上面方法，要求传入一个父类的原型作为参数，然后返回子类的原型。

不过，这样我们还是遗漏了一点东西——子类不只是继承父类的遗产，还拥有自己的东西。此外，原型继承并没有让子类继承父类的类成员与特权成员。这些我们还得手动添加，比如类成员，我们可以通过上面的extend方法，特权成员我们可以在子类的构造器中，通过apply实现。

```
function inherit(init, Parent, proto){
    function Son(){
        Parent.apply(this,argument);    //先继承父类的特权成员
        init.apply(this,argument);    //再执行自己的构造器
    }
    //由于Object.create可能是我们伪造的，因此避免使用第二个参数
    Son.prototype = Object.create(Parent.prototype,{});
    Son.prototype.toString = Parent.prototype.toString;//处理IE
Bug
    Son.prototype.valueOf = Parent.prototype.valueOf;//处理IE Bu
g
    Son.prototype.constructor = Son;    //确保构造器正常指向自身，而
不是Object
    extend(Son.prototype, proto);       //添加子类特有的原型成员
    extend(Son, Parent);                  //继承父类的类成员
    return Son;
}
```

下面我们做一组实验，测试一下实例的回溯机制。许多资料都说，但总是语焉不详。当我们访问对象的一个属性，那么它先找其特权成员，如果有同名的就返回，没有就找原型，再没有，找父类的原型……我们尝试把它的原型临时修改一下，看它的属性会变成哪一个！

```
function A() {}
A.prototype = {
  aa: 1
}
var a = new A;
console.log( a.aa);//1
//把它整个原型对象都换掉
A.prototype = {
  aa: 2
}
console.log(a.aa);//1，表示不受影响
//于是我们想到实例都有一个constructor方法，指向其构造器，
//而构造器上面正好有我们的原型，JavaScript引擎是不是通过该路线回溯属性呢
function B(){}
B.prototype = {
  aa: 3
}
a.constructor = B;
console.log( a.aa );//1 表示不受影响
```

因此类的实例肯定通过另一条通道进行回溯，翻看ecma规范可知每一个对象都有一个内部属性[[Prototype]]，它保存着当我们new它时构造器所引用的prototype对象。在标准浏览器与IE11里，它们暴露了一个叫__proto__属性来访问它。因此只要不动__proto__，上面的代码怎么动，a.aa始终坚定不移地返回1。我们再来看一下new操作时发生了什么事。

（1）创建一个空对象instance。

（2）instance.__proto__ = instanceClass.prototype。

（3）将构造器函数里面的this = instance。

（4）执行构造器里面的代码。

（5）判定有没有返回值，如果有，则判定返

回值的类型，如果类型为Object, Array等复合数据类型，就返回该对象，否则返回this（实例）。

于是有了下面结果。

```
function A() {
  console.log(this.__proto__.aa); //1
  this.aa = 2
}
A.prototype = {
  aa: 1
}
var a = new A;
console.log(a.aa); //2
a.__proto__ = {
  aa: 3
}
delete a.aa; //删掉特权属性，暴露原型链上的同名属性
console.log(a.aa); //3
```

有了`__proto__`，我们可以将原型继承设计得更简洁。我们还是拿上面的例子改一下来进行实验。

```
function A() {}
A.prototype = {
  aa: 1
}

function bridge() {};
bridge.prototype = A.prototype;
```

```
function B() {}
B.prototype = new bridge();
B.prototype.constructor = B;
var b = new B;
B.prototype.cc = function() {
  alert(3)
}
console.log(b.__proto__ == B.prototype);
//true 这个大家应该都没有疑问
console.log(b.__proto__.__proto__ === A.prototype);
//true  得到父类的原型对象
```

为什么呢？因为b.__proto__ .constructor为B，而B的原型是从bridge中得来的，而bridge. prototype = A.prototype。反过来，我们在定义时，让B.prototype.__proto__ = A.prototype，就能轻松实现两个类的继承。

目前，__proto__ 属性已列入ES6，因此可以通过if分支大胆使用它。

4.2　各种类工厂的实现

由于JavaScript是Brendan Eich“大牛人”在10天之内发明，但后来马上遇上浏览器大战，网景公司失势，JavaScript的后期改良升级就一直耽误了。其中继承的遗留问题是最严重的一个。当我们要复用某些功能时，第一个反应就是想到继承，然后才是组合。尽管Gof [1] 再三重申，组合优于继承，但继承这种代码复用手法，在绝大多数的语言直接可以使用类来实现，而组合则需要用设计模式自己搞。

由于JavaScript没有实现类，先行者折腾出好几种实现来模拟其他语言的类（如原型链继承、构造函数继承、组合继承、寄生继承、寄生组合继承……），它们都各有优劣。但我们将这些创建类的方法封装一下，供别人用时，就需要考虑到更多功能了。于是有了类工厂这一说，这个工厂专门是用来制作各种类，然后通过类再来制造我们实际使

用的实例对象。

由于主流框架的类工厂实现太依赖于它们庞杂的工具函数，而一个精巧的类工厂也不过百行左右，因此本章就不打算罗列Prototype.js、Mootools等的代码了，介绍另外一些不太出名但相当有水准的小库吧。

4.2.1 相当精巧的库—P.js

这是一个相当精巧的库，尤其在调用父类的同名方法时，它直接把父类的原型抛在你眼前，连_super也省了。

它的源码解读如下。

```
var P = (function(prototype, ownProperty, undefined) {

    function isObject(o) {
        return typeof o === 'object';
    }

    function isFunction(f) {
        return typeof f === 'function';
    }

```

```
    function BareConstructor() {};

    function P(_superclass /* = Object */ , definition) {
        //如果只传一个参数，没有指定父类
        if(definition === undefined) {
            definition = _superclass;
            _superclass = Object;
        }

        //C为我们要返回的子类，definition中的init为用户构造器

        function C() {
            var self = new Bare;
            console.log(self.init)
            if(isFunction(self.init)) self.init.apply(self, arg
uments);
            return self;
        }

        function Bare() { //这个构造器是为了让C不用new就能返回实例而设
的
        }
        C.Bare = Bare;
        //为了防止改动子类影响到父类，我们将父类的原型赋给一个中介者BareC
onstructor
        //然后再将这中介者的实例作为子类的原型
        var _super = BareConstructor[prototype] = _superclass[p
rototype];
        var proto = Bare[prototype] = C[prototype] = new BareCo
nstructor; //
        //然后C与Bare都共享同一个原型
        //最后修正子类的构造器指向自身
        proto.constructor = C;
        //类方法mixin,不过def对象里面的属性与方法糅杂到原型里面去
        C.mixin = function(def) {
            Bare[prototype] = C[prototype] = P(C, def)[prototyp
e]; //Bare[prototype] =
            return C;
        }
        //definition最后延迟到这里才起作用
        return(C.open = function(def) {
            var extensions = {};
            //definition有两种形态
            //如果是函数，那么子类原型、父类原型、子类构造器、父类构造传
进去，
```

```
            //如果是对象则直接置为extensions
            if(isFunction(def)) {
                extensions = def.call(C, proto, _super, C, _sup
erclass);
            } else if(isObject(def)) {
                extensions = def;
            }
            //最后混入子类的原型中
            if(isObject(extensions)) {
                for(var ext in extensions) {
                    if(ownProperty.call(extensions, ext)) {
                        proto[ext] = extensions[ext];
                    }
                }
            }
            //确保init为一个函数
            if(!isFunction(proto.init)) {
                proto.init = _superclass;
            }

            return C;
        })(definition);

        //这里为一个自动执行函数表达式，相当于
        //C.open = function(){/*....*/}
        //C.open(definition)
        //return C;
        //换言之，返回的子类存在3个类成员，Base, mixin, open

    }

    return P; //暴露到全局
})('prototype', ({}).hasOwnProperty);
```

我们尝试创建一个类。

```
var Animal = P(function(proto, superProro) {
  proto.init = function(name) { //构造函数
    this.name = name;
  };
```

```
  proto.move = function(meters) { //原型方法
    console.log(this.name + " moved " + meters + "m.");
  }
});
var a = new Animal("aaa")
var b = Animal("bbb");//无"new"实例化

a.move(1)
b.move(2)
```

此外，我们还可以使用以下更简洁的定义方式。

```
var Snake = P(Animal, function(snake, animal) {
  snake.init = function(name, eyes) {
    animal.init.call(this, arguments); //调用父类构造器
    this.eyes = 2;
  }
  snake.move = function() {
    console.log("Slithering...");
    animal.move.call(this, 5); //调用父类的同名方法
  };
});
var s = new Snake("snake", 1);
s.move();
console.log(s.name);
console.log(s.eyes);
```

下面是私有属性的演示，由于放在函数体内集中定义，因此安全可靠！

```
var Cobra = P(Snake, function(cobra) {
```

```
  var age = 1; //私有属性
  //这里还可以编写私有方法
  cobra.glow = function() { //长大
    return age++;
  }
});
var c = new Cobra("cobra");
console.log(c.glow()); //1
console.log(c.glow()); //2 又长一岁
console.log(c.glow()); //3 又长一岁
```

此外它还提供了两个类方法，mixin用于再次添加新的原型成员，open的作用同mixin，但显然它适合于重写父类的方法（在子类方法内部重用父类方法），同时，也可以添加新的私有属性。open这个命名显然是受ruby影响，意为重新打开类，修改其原型。

4.2.2　JS.Class

其仓库地址：

https://github.com/RubyLouvre/JS.Class

从它的设计来看，它是师承Dean Edwards“大

牛人”的Base2，相似的类工厂实现还有mootools和第4.2.3节的simple-inheritance。它是通过父类构造器的extend方法来产生自己的子类，里面存在一个开关，防止在生成类时无意执行construct方法。比如Base2的base2.__prototyping和mootools的klass.$prototyping。它创建子类时，也不通过中间的函数来断开双方的原型链，而是使用父类的实例来做子类的原型，这点实现得非常精巧。

源码解读如下。

```
var JS = {
    VERSION: '2.2.1'
};

JS.Class = function(classDefinition) {

    //返回目标类的真正构造器
    function getClassBase() {
        return function() {
            //它在里面执行用户传入的构造器construct
            //preventJSBaseConstructorCall是为了防止在createClass
Definition辅助方法中执行
            //父类的construct
            if (typeof this['construct'] === 'function' && prev
entJSBaseConstructorCall
            === false) {
                this.construct.apply(this, arguments);
            }
        };
```

```
    }
    //为目标类添加类成员与原型成员
    function createClassDefinition(classDefinition) {
        //此对象用于保存父类的同名方法
        var parent = this.prototype["parent"] || (this.prototyp
e["parent"] = {});
        for (var prop in classDefinition) {
            if (prop === 'statics') {
                for (var sprop in classDefinition.statics) {
                    this[sprop] = classDefinition.statics[sprop
];
                }
            } else {
    //为目标类添加原型成员，如果是函数，那么检测它还没有同名的超类方法，如
果有则进入以下分支
                if (typeof this.prototype[prop] === 'function')
 {
                   var parentMethod = this.prototype[prop];
                   parent[prop] = parentMethod;
                }
                this.prototype[prop] = classDefinition[prop];
            }
        }
    }

    var preventJSBaseConstructorCall = true;
    var Base = getClassBase();
    preventJSBaseConstructorCall = false;

    createClassDefinition.call(Base, classDefinition);

    //用于创建当前类的子类
    Base.extend = function(classDefinition) {

        preventJSBaseConstructorCall = true;
        var SonClass = getClassBase();
        SonClass.prototype = new this();//将一个父类的实例当作子类的
原型
        preventJSBaseConstructorCall = false;

        createClassDefinition.call(SonClass, classDefinition);
        SonClass.extend = this.extend;

        return SonClass;
    };
```

```
    return Base;
};
```

创建一个Animal类与一个Dog子类。

```
var Animal = JS.Class({
  construct: function(name) {
    this.name = name;
  },
  shout: function(s) {
    console.log(s);
  }
});

var animal = new Animal();
animal.shout('animal'); // animal
var Dog = Animal.extend({
  construct: function(name, age) {
    //调用父类构造器
    this.parent.construct.apply(this, arguments);
    this.age = age;
  },
  run: function(s) {
    console.log(s);
  }
});
var dog = new Dog("dog", 4);
console.log(dog.name);
dog.shout("dog"); // dog
dog.run("run"); // run
```

演示静态成员的实现如下。

```
var Shepherd = Dog.extend({
  statics: { //静态成员
```

```
    TYPE: "Shepherd"
  },
  run: function() { //方法链，调用超类同名方法
    this.parent.run.call(this, "fast");
  }
});
console.log(Shepherd.TYPE); //Shepherd
var shepherd = new Shepherd("shepherd", 5);
shepherd.run(); //fast
```

JS.Class虽然功能稍微薄弱些，但简洁得惊人。我们可以在它的基础上学习Base2和mootools的实现。

4.2.3 simple-inheritance

作者为大名鼎鼎的**John Resig**，项目直接放在他的一篇博文中。

特点是方法链的实现非常优雅，简洁！

源码解读如下。

```
(function() {
    // /xyz/.test(function(){xyz;})是用于判定函数的toString是否能暴露里面的实现
    // 因为Function.prototype.toString没有做出强制规定如何显示自身，
```

```
根据浏览器实现而定
    // 如果里面能显示内部内容，那么我们就使用 /\b_super\b/来检测函数里面有没有.super语句
    // 当然这个也不很充分，只是够用的程度；否则就返回一个怎么也返回true的正则
    // 比如一些古老版本的Safari、Mobile Opera与 Blackberry浏览器，无法显示函数体的内容
    // 就需要用到后面的正则
    var initializing = false, fnTest = /xyz/.test(function() {
        xyz;
    }) ? /\b_super\b/ : /.*/;

    // 所有人工类的基类
    this.Class = function() {
    };

//这是用于生成目标类的子类
    Class.extend = function(prop) {
        var _super = this.prototype;//保存父类的原型

        //阻止init被触发
        initializing = true;
        var prototype = new this();//创建子类的原型
        //重新打开，方便真实用户可以调用init
        initializing = false;
        //将prop里的东西逐个复制到prototype,如果是函数将特殊处理一下
        //因为复制过程中可能掩盖了超类的同名方法，如果这个函数里面存在_super的字样，就笼统地
        //认为它需要调用父类的同名方法，那么我们需要重写当前函数
        //重写函数运用了闭包，因此fnTest正则检测可以减少我们重写方法的个数，
        //因为不是每个同名函数都会向上调用父类方法
        for (var name in prop) {
            prototype[name] = typeof prop[name] === "function" &&
                    typeof _super[name] === "function" && fnTest.test(prop[name]) ?
                    (function(name, fn) {
                        return function() {
                            var tmp = this._super;//保存到临时变量中
                        //当我们调用时，才匆匆把父类的同方法覆写到_super里
                        this._super = _super[name];
                        //然后才开始执行当前方法（这时里面的this._sup
```

```
er已被重写），得到想要的效果
                        var ret = fn.apply(this, arguments);
                        //还原this._super
                        this._super = tmp;
                        //返回结果
                        return ret;
                    };
            })(name, prop[name]) :
            prop[name];
        }

        // 这是目标类的真实构造器
        function Class() {
        // 为了防止在生成子类的原型（new this()）时触发用户传入的构造器i
nit
        // 使用initializing进行牵制
            if (!initializing && this.init)
                this.init.apply(this, arguments);
        }

        //将修改好的原型赋值
        Class.prototype = prototype;

        // 确保原型上constructor正确指向自身
        Class.prototype.constructor = Class;

        //添加extend类方法，生于生产它的子类
        Class.extend = arguments.callee;

        return Class;
    };
})();
```

创建一个Animal类与一个Dog子类。

```
var Animal = Class.extend({
  init: function(name) {
    this.name = name;
  },
  shout: function(s) {
```

```
    console.log(s);
  }
});

var animal = new Animal();
animal.shout('animal'); // animal
var Dog = Animal.extend({
  init: function(name, age) {
    //调用父类构造器
    this._super.apply(this, arguments);
    this.age = age;
  },
  run: function(s) {
    console.log(s);
  }
});
var dog = new Dog("dog", 4);
console.log(dog.name); //dog
dog.shout("xxx"); // xxx
dog.run("run"); // run
console.log(dog instanceof Dog && dog instanceof Animal);//true
```

顺便一提，simple-inheritance的老师有两个，Base2的继承系统与Prototype.js的方法链系统。Prototype.js与simple-inheritance都是对函数toString进行反编译，看里面有没有_super或$super的字眼才决定是否重写。不同的是Prototype.js只检测方法的参数列表，因此杂质更少，更加可靠！

下面是Prototype.js方法链演示。

```
var Person = Class.create();
Person.prototype = {
  initialize: function(name) {
    this.name = name;
  },
  say: function(message) {
    return this.name + ': ' + message;
  }
};

var guy = new Person('Miro');
console.log(guy.say('hi')); // "Miro: hi"
//创建子类
var Pirate = Class.create(Person, {
  say: function($super, message) { //注意这里的传参，$super为超类的同名方法
    return $super(message) + ", yarr!"; //这需要外科手术般的闭包来实现
  }
});

var john = new Pirate('Long John');
console.log(john.say('good bye')) //Long John: good bye, yarr!
```

当然Prototype.js这样做有个缺憾，导致定义时与使用时的参数不一样。对于大多数用户来说，实现并不是他们所关心的，保持简洁优雅的接口才是重点。如果翻看jQuery UI的源码会发现，它的widget类工厂已经把方法链设计得登峰造顶了。它会在函数的this对象添加两个临时方法，_super相当于simple-inheritance的_super，它的参数需要用户逐

个转手传递；_superApply是_super的强化版，因为如果外围方法是不确定的，那么你也没法为_super传参，因此它只需用户丢个arguments对象进去！

4.2.4　体现JavaScript灵活性的库——def.js

其仓库地址：https://github.com/RubyLouvre/def.js

如果有什么库最能体现JavaScript的灵活性，此库肯定名列前茅。它试图在形式上模拟Ruby那种继承，让学过Ruby的人一眼就看到哪个是父类，哪个是子类。

下面就是Ruby的继承示例。

```
class Child < Father
  #略
end
```

def.js能做如下这个程度。

```
def("Animal")({
  init: function(name) {
    this.name = name;
  },
  speak: function(text) {
    console.log("this is a " + this.name);
  }
});
var animal = new Animal("Animal");

console.log(animal.name)

def("Dog") < Animal({
  init: function(name, age) {
    this._super(); //魔术般地调用父类
    this.age = age;
  },
  run: function(s) {
    console.log(s)
  }
});
var dog = new Dog("wangwang");
console.log(dog.name); //wangwang

//在命名空间对象上创建子类
var namespace = {}
def(namespace, "Shepherd") < Dog({
  init: function() {
    this._super();
  }
});

var shepherd = new namespace.Shepherd("Shepherd")
console.log(shepherd.name);
```

由于涉及的魔术比较多，我逐个分解一下。

第一个是**curry** 的运用，在`def("Animal")`之后它还能直接添加括号，说明它是一个函数。而此时真正的类已经创建出来，window.Animal就访问到这个类。

第二个是使用<运算符对 **valueOf** 的劫持。其实原项目是用`<<`，不过换成+、−也行，但要保证与Ruby的拟态，我还是推荐用`<`。`<`操作符的目的是强制两边计算自身，从而调用自己的`valueOf`方法。def.js就是通过重写了父类与子类定义的valueOf实现在某个作用域中偷偷地进行原型继承，如图4-2所示。



图4-2

```
var a = {valueOf:function(){
  console.log("aaaaaaa")
}}, b = {valueOf:function(){
  console.log("bbbbbbb")
}}
a < b
```

由于操作符两边都是函数，那么我们能做更多的事！

```
function def(name) {
  console.log("def(" + name + ") called")
  var obj = {
    valueOf: function() {
      console.log(name + " (valueOf)")
    }
  }
  return obj
}
def("Dog") < def("Animal");
```

第三个是**arguments.callee.caller** 的运用 。大家看一下Dog的构造函数，里面只有一句this._super()，没有传参，但它依然能调用到它的父类构造器，并把arguments塞进

去。**arguments.callee** 就是指_super这个函数，caller就是init这个函数，然后我们访问**caller.arguments**， 就得到"wangwang"这个传参了。因此_super比simple-inheritance智能多了，就像Java的super关键字那样，摆在那里自行干活。同时_super不但能自动调用父类的构造器，同名超类方法的实现也由它一手打包。

下面是源码解读。

```
(function(global) {
  //deferred是整个库中最重要的构件，扮演3个角色
  //1 def("Animal")时就是返回deferred，此时我们可以直接接括号对原型进行扩展
  //2 在继承父类时 < 触发两者调用valueOf，此时会执行deferred.valueOf里面的逻辑
  //3 在继承父类时，父类的后面还可以接括号（废话，此时构造器当普通函数使用），当作传送器，
  //  保存着父类与扩展包到_super,_props
  var deferred;

  function extend(source) { //扩展自定义类的原型
    var prop, target = this.prototype;

    for(var key in source)
    if(source.hasOwnProperty(key)) {
      prop = target[key] = source[key];
      if('function' == typeof prop) {
        //在每个原型方法上添加两个自定义属性，保存其名字与当前类
        prop._name = key;
        prop._class = this;
```

```
      }
    }

    return this;
  }
  // 一个中介者，用于切断子类与父类的原型连接
  //它会像DVD+R光盘那样被反复擦写

  function Subclass() {}

  function base() {
    // 取得调用this._super()这个函数本身，如果是在init内，那么就是当前
类
    //http://larryzhao.com/blog/arguments-dot-callee-dot-caller
-bug-in-internet-explorer-9/
    var caller = base.caller;
    //执行父类的同名方法，有两种形式，一是用户自己传，二是智能取当前函数的
参数
    return caller._class._super.prototype[caller._name].apply(t
his, arguments.length ?
    arguments : caller.arguments);
  }

  function def(context, klassName) {
    klassName || (klassName = context, context = global);
    //偷偷在给定的全局作用域或某对象上创建一个类
    var Klass = context[klassName] = function Klass() {
        if(context != this) { //如果不使用new 操作符，大多数情况下co
ntext与this都为window
          return this.init && this.init.apply(this, arguments);
        }
        //实现继承的第二步，让渡自身与扩展包到deferred
        deferred._super = Klass;
        deferred._props = arguments[0] || {};
      }

      //让所有自定义类都共用同一个extend方法
      Klass.extend = extend;

    //实现继承的第一步，重写deferred，乍一看是刚刚生成的自定义类的扩展函
数
    deferred = function(props) {
      return Klass.extend(props);
    };
```

```
    // 实现继承的第三步，重写valueOf，方便在def("Dog") < Animal({})执行它
    deferred.valueOf = function() {

      var Superclass = deferred._super;

      if(!Superclass) {
        return Klass;
      }
      // 先将父类的原型赋给中介者，然后再将中介者的实例作为子类的原型
      Subclass.prototype = Superclass.prototype;
      var proto = Klass.prototype = new Subclass;
      // 引用自身与父类
      Klass._class = Klass;
      Klass._super = Superclass;
      //一个小甜点，方便人们知道这个类叫什么名字
      Klass.toString = function() {
        return klassName;
      };
      //强逼原型中的constructor指向自身
      proto.constructor = Klass;
      //让所有自定义类都共用这个base方法，它是构成方法链的关系
      proto._super = base;
      //最后把父类后来传入的扩展包混入子类的原型中
      deferred(deferred._props);
    };

    return deferred;
  }

  global.def = def;
}(this));
```

它的实现非常巧妙。这要对def(“Dog”) < Animal({})这一行的代码各个部分的执行顺序有充分的了解。无疑左边的def会先执行，重新擦写了

deferred与deferred.valueOf。然后是父类Animal作为普通函数接受子类的扩展包，扩展包与父类也在这时偷偷附加到deferred上。最后是中间的操作符触发deferred.valueOf，完成继承！

当然也有美中不足的地方，就是利用了caller这个被废弃的属性。在ES5的严格模式下，它是不可用的，会导致继承系统瘫痪！这个修改也很简单，直接参考jQuery UI的那部分就行了，只是少了一些智能化。

4.3　进击的属性描述符

ES5最瞩目的升级是为对象引入属性描述符，就像新事件对旧世界的颠覆性的冲击。在过去，我们知道原生对象的某些属性不能删除或遍历，但不能深究下去，也无法改变什么。现在属性描述符给了我们一个完整的答案，并且能让我们用它做更多的事情，如实现不可变对象或者实现avalon这样具有双向绑定能力的vm对象。

属性描述符让我们对属性有了更精细的控制，比如说这个属性是否可以修改、是否可以在for in循环中枚举出来、是否可以删除。这些新增的API都集中定义在Object下，基本上除了Object.keys这个方法外，旧版本IE都无法模拟其他新API。

Object下总供添加以下几种新方法。

- Object.keys
- Object.getOwnPropertyNames
- Object.getPrototypeOf
- Object.defineProperty
- Object.defineProperties
- Object.getOwnPropertyDescriptor
- Object.create
- Object.seal
- Object.freeze
- Object.preventExtensions
- Object.isSealed
- Object.isFrozen *
- Object.isExtensible

其中除**Object.keys** 外，旧版本IE都无法模拟这些新API。旧版式的标准浏览器，可以用`__proto__`实现**Object.getPrototypeOf**，结合`__defineGetter__`与`__defineSetter__`来模拟

Object.defineProperty 。

Object.keys 用于收集当前对象的可遍历属性（不包括原型链上的），以数组形式返回。这个我在第2章中已经给出兼容函数。

Object.getOwnPropertyNames 用于收集当前对象不可遍历属性与可遍历属性（不包括原型链上），以数组形式返回。

```
var obj = {
  aa: 1,
  toString: function() {
    return "1"
  }
}
if(Object.defineProperty && Object.seal) {
  Object.defineProperty(obj, "name", {
    value: 2
  })
}
console.log(Object.getOwnPropertyNames(obj));//["aa", "toString
", "name"]
console.log(Object.keys(obj));//["aa", "toString"]

function fn(aa, bb) {};
console.log(Object.getOwnPropertyNames(fn));//["prototype","len
gth","name","arguments","caller"]
console.log(Object.keys(fn));//[]
var reg = /\w{2,}/i

console.log(Object.getOwnPropertyNames(reg));//["lastIndex","so
urce","global","ignoreCase","mnltiline","sticky"]
```

```
console.log(Object.keys(reg));//[]
```

Object.getPrototypeOf 返回参数对象的内部属性`[[Prototype]]`，它在标准浏览器中一直使用一个私有属性**proto** 获取（IE9、IE10和Opera没有）。

需要补充一下，Object的新API（除了Object.create）有个统一的规定，要求第一个参数不能为数字、字符串、布尔、null、undefined这五种的字面量，否则抛出一个TypeError异常。

```
console.log(Object.getPrototypeOf(function() {}) == Function.prototype); //true
console.log(Object.getPrototypeOf({}) === Object.prototype); //true
```

Object.defineProperty暴露了属性描述的接口，之前许多内建属性都是由JavaScript引擎在水下操作。比如说for in循环为何不能遍历出函数的

arguments、length、name等属性名，delete window.a为何返回false，这些现象终于有个解释。它一共涉及6个可组合的配置项：是否可重写writable；当前值value；读取时内部调用的函数get；写入时内部调用函数set；是否可以遍历enumerable；是否可让人家再次改动这些配置项configurable。比如我们随便写个对象：

```
var obj = { x : 1 };
```

有了属性描述符，我们就清楚它在底下做的更多细节，它相当于以下ES5代码：

```
var obj = Object.create(Object.prototype,
  { x : {
    value : 1,
    writable : true,
    enumerable : true,
    configurable : true
  }}
)
```

如果对比ES3与ES5，就很快明白，曾经的

[[ReadOnly]]、[[DontEnum]]、[[DontDelete]]改换成[[Writable]]、[[Enumerable]]、[[Configurable]]了。

这6个配置项将原有的本地属性拆分为两组，数据属性与访问器属性。我们之前的方法可以像数据属性那样定义，如图4-3所示。

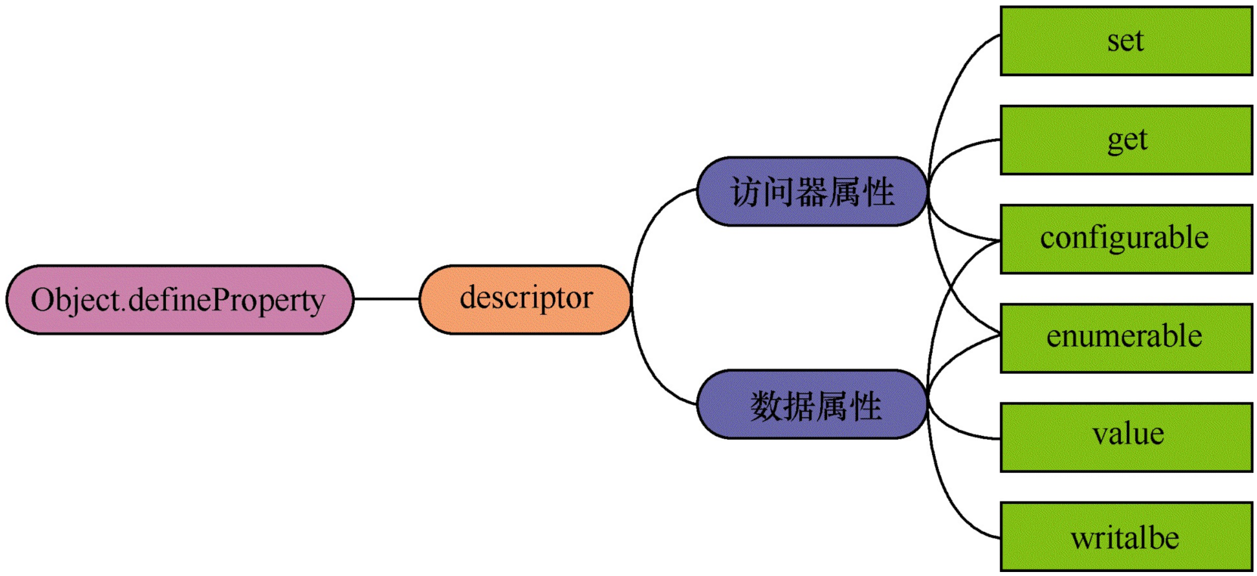


图4-3

ES3时代，我们自定义类的属性可以统统看作是数据属性。

像DOM中的元素节点的innerHTML、innerText、cssText，数组的length则可归为访问器属性，对它们赋值时不是单纯的赋值，还会引发元素其他功能的触发，而取值也不一定直接返回我们之前给予的值。

```
var obj = {};
Object.defineProperty(obj, "a", {
  value: 37,
  writable: true,
```

```
  enumerable: true,
  configurable: true
});

console.log(obj.a);//37
obj.a = 40;
console.log(obj.a);//40
var name = "xxx"
for(var i in obj){
    name = i
}
console.log(name);//a

Object.defineProperty(obj, "a", {
  value: 55,
  writable: false,
  enumerable: false,
  configurable: true
});

console.log(obj.a);//55
obj.a = 50;
console.log(obj.a);//55
name = "b";
for(var i in obj){
    name = i
}
console.log(name);//b

var value = "RubyLouvre";
Object.defineProperty(obj, "b", {
  set: function(a){
    value = a;
  },
  get: function(){
    return value + "!"
  }
});

console.log(obj.b);//RubyLouvre!
obj.b = "bbb";
console.log(obj.b);//bbb!

var obj = Object.defineProperty( {} , 'a', {
  value: "aaa"
```

```
});
delete obj.a;//configurable默认为false，此属性不能删除
console.log(obj.a);//aaa
```

但这东西各浏览器也有差异。

```
var arr = [];
//添加一个属性，但由于是数字字面量，它又会作为数组的第一个元素
Object.defineProperty(arr, '0', {value : "零"});
Object.defineProperty(arr, 'length', {value : 10});
//删除第一个元素，但由于length的writable在上面被我们设置为false(不写默认为false)，因此改不了。
arr.length = 0 ;
alert([arr.length, arr[0]]);//正确应该输出"1，零"
//IE9、IE10："1,零"
//Firefox4～Firefox19：抛内部错误，说当前不支持定义length属性
//Safari5.0.1："0, "，第二值应该是undefined，说明它忽略了writable为false的默认设置，让arr.length把第一个元素删掉了
//Chrome14-："0,零"，估计后面的"零"是作为属性打印出来，chrome24与标准保持一致。
```

此外，defineProperty的第三个参数配置对象好像没有使用hasOwnProperty进行取值，导致一旦Object.prototype被污染，就很容易程序崩溃。这情况好像所有现代浏览器都遇到了。

```
Object.prototype.set = undefined
var obj = {};
Object.defineProperty(obj, "aaa", { value: "OK" });
//TypeError: property descriptor's getter field is neither unde
```

```
fined nor a function
```

或者：

```
Object.prototype.get = function(){};
var obj = {};
Object.defineProperty(obj,  "aaa", { value: "OK" });
//TypeError: property descriptors must not specify a value or b
e writable when a getter or setter has been specified
```

如果真的碰巧让你撞上这事，唯有自力更生了。

```
function hasOwn(obj, key) {
  return Object.prototype.hasOwnProperty.call(obj, key);
}
function defineProperty(obj, key, desc) {
//创建一个纯空对象，不继承Object.prototype，跳过那些粗糙的for in遍历BUG
  var d = Object.create(null);
  d.configurable = hasOwn(desc,"configurable");
  d.enumerable = hasOwn(desc, "enumrable");
  if (hasOwn(desc, "value")) {
    d.writable = hasOwn(desc, "writable")
    d.value = desc.value;
  } else {
    d.get = hasOwn(desc, "get") ? desc.get : undefined;
    d.set = hasOwn(desc, "set") ? desc.set : undefined;
  }
  return Object.defineProperty(obj, key, d);
}
var obj = {};
defineProperty(obj, "aaa", { value: "OK" });//save!
```

在标准浏览器中，如果不支持Object.defineProperty，我们可以勉强模拟它出来。

```
if(typeof  Object.defineProperty!=='function'){
    Object.defineProperty = function(obj, prop, desc) {
        if ('value' in desc) {
            obj[prop] = desc.value;
        }
        if ('get' in desc) {
            obj.__defineGetter__(prop, desc.get);
        }
        if ('set' in desc) {
            obj.__defineSetter__(prop, desc.set);
        }
        return obj;
    };
}
```

Object.defineProperties 就是Object.defineProperty的加强版，它能一下子处理多个属性。因此如果你能模拟Object.defineProperty，它就不是问题。

```
if(typeof  Object.defineProperties!=='function'){
    Object.defineProperties = function(obj, descs) {
        for (var prop in descs) {
            if (descs.hasOwnProperty(prop)) {
                Object.defineProperty(obj, prop, descs[prop]);
            }
```

```
        }
        return obj;
    };
}
```

使用示例：

```
var obj = {};
Object.defineProperties(obj, {
  "value": {
    value: true,
    writable: false
  },
  "name": {
    value: "John",
    writable: false
  }
});
var a = 1;
for(var p in obj) {
  a = p;
}
console.log(a);//1
```

Object.getOwnPropertyDescriptor 是用于获得某对象的本地属性的配置对象，其中configurable、enumerable肯定包含其中，视情况再包含value、writable或set、get。

```
var obj = {},=value = 0
```

```
Object.defineProperty(obj, "aaa", {
    set: function(a) {
      value = a;
    },
    get: function() {
      return value
    }
});
//一个包含set, get, configurable, enumerable的对象
console.log(Object.getOwnPropertyDescriptor(obj, "aaa"));
console.log(typeof obj.aaa);//number
console.log(obj.hasOwnProperty("aaa"));//true

(function() {
//一个包含value, writable, configurable, enumerable的对象
  console.log(Object.getOwnPropertyDescriptor(arguments, "lengt
h"))
})(1, 2, 3);
```

由于属性在现代浏览器划分两阵营了，如果我们想把一个对象的成员赋给另一个对象，原来的mixin就会捉襟见肘。这时 **Object.getOwnPropertyDescriptor** 就派上用场了。

```
function mixin(receiver, supplier) {
    if (Object.getOwnPropertyDescriptor) {
        Object.keys(supplier).forEach(function(property) {
            Object.defineProperty(receiver, property, Object.ge
tOwnPropertyDescriptor (supplier, property));
        });
    } else {
        for (var property in supplier) {
            if (supplier.hasOwnProperty(property)) {
                receiver[property] = supplier[property];
            }
```

```
        }
    }
}
```

Object.create用于创建一个子类的原型，第一个参数为父类的原型，第二个是子类另外要添加的属性的配置对象。如果我们能模拟Object.defineProperties，它也能模拟得到。

```
if(typeof  Object.create !== 'function') {
  Object.create = function(prototype, descs) {
    function F() {}
    F.prototype = prototype;
    var obj = new F();
    if(descs != null) {
      Object.defineProperties(obj, descs);
    }
    return obj;
  };
}
```

这个API的出现，其实有着另一重深厚的意味——预示着JavaScript着手抗拒它早期失败的设计。在JavaScript中，每一个函数都可以当作构造函数，所以我们需要区分普通的函数调用和构造函数调

用；我们一般使用new 关键字来进行区别。然而，这样就破坏了JavaScript中的函数式特点，因为new是一个关键字而不是函数。因而函数式的特点无法和对象实例化一起使用。此外，new 关键字掩盖了JavaScript中真正的原型继承，使得它更像是基于类的继承。就像Raynos说的：new 是JavaScript为了获得流行度而加入与Java类似的语法时期留下来的一个残留物，JavaScript是一个源于Self的基于原型的语言。然而，为了市场需求，Brendan Eich把它当成Java的小兄弟推出。并且我们当时把JavaScript当成Java的一个小兄弟，就像在微软语言家庭中Visual Basic相对于C++一样。

这个设计决策导致了new的问题。当人们看到JavaScript中的new关键字，他们就想到类，然后当他们使用继承时就傻了。就像Douglas Crockford说的：这个间接的行为是为了使传统的程序员对这门语言更熟悉，但是却失败了，就像我们看到的很少

有Java程序员选择了JavaScript。JavaScript的构造模式并没有吸引传统的人群，它也掩盖了JavaScript基于原型的本质。结果就是，很少的程序员知道如何高效的使用这门语言。

Object.create的出现，允许我们从零开始创建一个对象，像下面这样。

```
var object = Object.create(null) ;
```

这种对象，我们称之为纯空对象。没有toString，也没有valueOf，空空荡荡的，在Object.prototype被污染或极需节省内存的情况下非常有用。它是一种超轻量的JavaScript对象。在firebug下，它与普通JavaScript对象是以不同的形式来表示，如图4-4所示。



图4-4

Object.create也能让我们基于某个现有的对象，创建一个新对象。这种行为，我们又称之为创建子类 。

```
function Animal(name) {
  this.name = name
}
Animal.prototype.getName = function() {
  return this.name;
}

function Dog(name, age) {
  Animal.call(this, name);
  this.age = age;
}
Dog.prototype = Object.create(Animal.prototype, {
  getAge: {
    value: function() {
      return this.age;
    }
  },
  setAge: {
    value: function(age) {
      this.age = age;
    }
  }
});

var dog = new Dog("dog", 4);
console.log(dog.name);//dog
dog.setAge(6);
console.log(dog.getAge());//6
```

Object.preventExtensions，它是3个封锁对象中修改的API中程度最轻的那个，就是阻止添加本地属性，不过如果本地属性被删除了，也无法再加回来。以前JavaScript对象的属性都是随意添加、删除、修改其值，如果它的原型改动，我们访问它还会有“意外之喜”。

```
var a = {
  aa: "aa"
};
Object.preventExtensions(a)
a.bb = 2;
console.log(a.bb);        //undefined 添加本地属性失败
a.aa = 3;
console.log(a.aa);        //3  允许它修改原有属性
delete a.aa;
console.log(a.aa);         //undefined 但允许它删除已有属性
Object.prototype.ccc = 4;
console.log(a.ccc);        //4  不能阻止它增添原型属性
a.aa = 5;
console.log(a.bb);        //undefined
```

Object.seal 比**Object.preventExtensions** 更过分，它不准删除已有的本地属性。内部实现就是遍历一下，把每个本地属性的configurable改为false。

```
var a = {
```

```
  aa: "aa"
};
Object.seal(a)
a.bb = 2;
console.log(a.bb);        //undefined 添加本地属性失败
a.aa = 3;
console.log(a.aa);        //3    允许它修改已有属性
delete a.aa;
console.log(a.aa);        //3   但不允许它删除已有属性
```

Object.freeze 无疑是最专制的（因此有人说过程式程序很专制，OO程序则自由些，显然道格拉斯主导的ECMA262V5想把JavaScript引向前者），它连原有本地属性也不让修改了。内部实现就是遍历一下，把每个本地属性的writable也改为false。

```
var a = {
  aa: "aa"
};
Object.freeze(a)
a.bb = 2;
console.log(a.bb);        //undefined 添加本地属性失败
a.aa = 3;
console.log(a.aa);        //aa    允许它修改已有属性
delete a.aa;
console.log(a.aa);        //aa   但不允许它删除已有属性

Object.isExtensible(object);
Object.isSealed(object);
Object.isFrozen(object);
```

判定一个对象是否被锁定。锁定，意味着无法扩展。如果一个对象被冻结了，它肯定被锁定，也肯定无法扩展新本地属性了。

4.4　真类降临

ES6中的Classes是在JavaScript现有的原型继承的基础上引入的一种“语法糖”。Class语法并没有引入一种新的继承模式。它为对象创建和继承提供了更清晰、易用的语法。并且，有了自带的原生类，我们就不需要用五花八门的方式来模拟类，在代码维护上是不可估量的！如图4-5和图4-6所示。

JavaScript自从被网景发明后，为了维护其正统性，它将JavaScript提交给ECMA委员会来负责标准化。由于刚好是第262号标准，所以便叫做ECMA262。又因为Java的版权在Java的公司Sun手里面，便指使ECMA将JavaScript的正式名称改为ECMAScript，简称ES。

1999年，制定了ES3的规范，将近10年时间语言层面没有大的改动。而到了2008年结尾，筹划了

好久的ES4由于加入改动过大，没有成功。于是删删减减，推出了更少改动，更友好的ES5。

2011年对ES5进修了一次修订，一直到2015年6月，ES6的规范发布了，并规定以后的ES规范以年份命名，每年发布一个小版本，再不会像ES6一下发布很多功能了，如图4-7所示。



图4-5

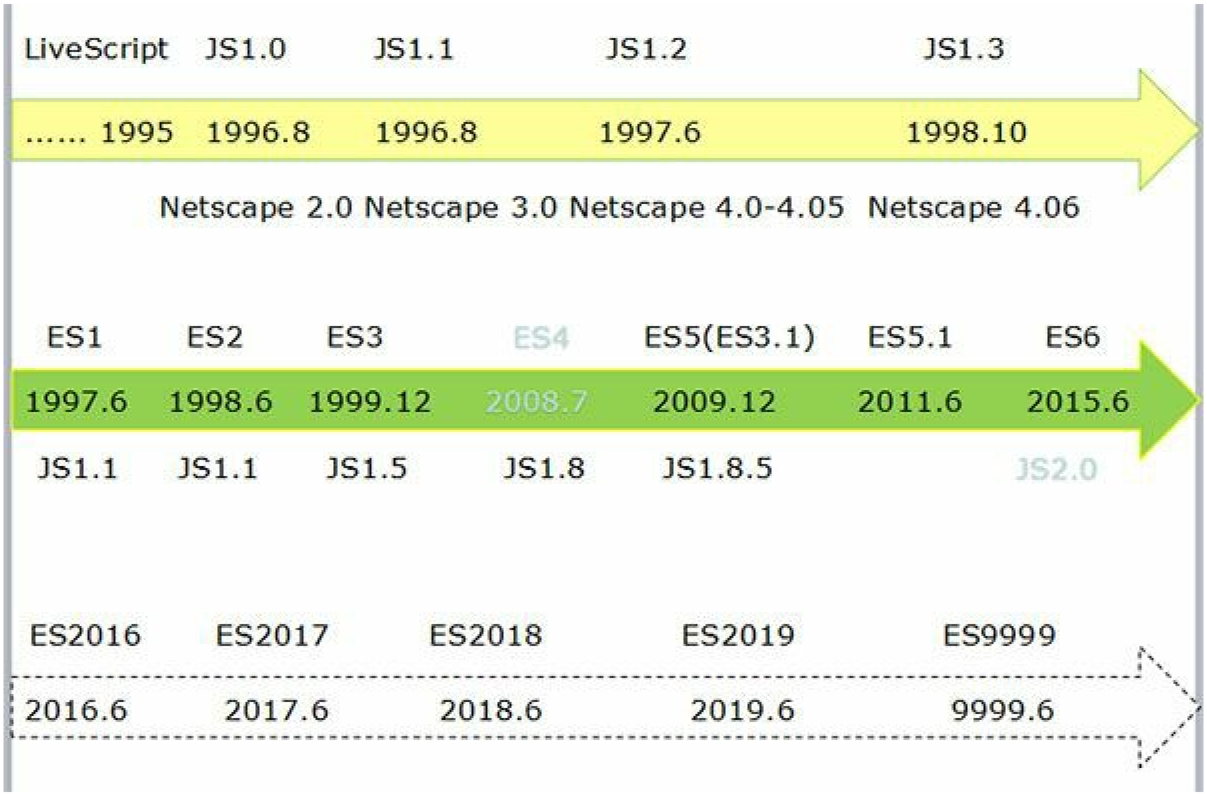

LiveScript
JS1.0
JS1.1
JS1.2
JS1.3
...... 1995
1996.8
1996.8
1997.6
1998.10
Netscape 2.0
Netscape 3.0
Netscape 4.0-4.05
Netscape 4.06
ES1
ES2
ES3
ES4
ES5(ES3.1)
ES5.1
ES6
1997.6
1998.6
1999.12
2008.7
2009.12
2011.6
2015.6
JS1.1
JS1.1
JS1.5
JS1.8
JS1.8.5
JS2.0
ES2016
ES2017
ES2018
ES2019
ES9999
2016.6
2017.6
2018.6
2019.6
9999.6

图4-6



图4-7

在ES6中，我们统一用class关键字来创建一个类，constructor关键字定义构造函数，用extends关键字来实现继承，super来实现调用父类方法。

我们先对比一下ES3与ES6是如何实现一个类的。

```
function Point(x,y){
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
```

```
}
//===========
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
```

ES3不出所料的是动用prototype对象来模拟，这与传统的面向对象语言有很大的出入，很容易让新学习这门语言的程序员感到困惑。

ES6的代码则接近传统语言的写法，引入了class关键字，并且有constructor构造方法，this关键字则代码实例对象。Point类除了构造方法，还定义了一个toString方法。注意，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。

ES6的类，完全可以看作构造函数的另一种写法。

```
class Point{
  // ...
}

typeof Point // "function"
Point === Point.prototype.constructor // true
```

上面代码表明，类的数据类型就是函数，类本身就指向构造函数。

构造函数的prototype属性，在ES6的类上面继续存在。事实上，类的所有方法都定义在类的prototype属性上面。

```
class Point {
  constructor(){
    // ...
  }

  toString(){
    // ...
  }

  toValue(){
    // ...
  }
```

```
}

// 等同于

Point.prototype = {
  toString: function(){},
  toValue: function(){}
}
```

在类的实例上面调用方法，其实就是调用原型上的方法。

```
class B {}
let b = new B();

b.constructor === B.prototype.constructor // true
```

上面代码中，b是B类的实例，它的constructor方法就是B类原型的constructor方法。由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。不过，在类里面定义的方法，都是不可遍历的，这一点与ES5的行为不一致。

```
Object.keys(Point.prototype)
// []
```

```
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]
```

constructor 方法 是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

当我们要通过类生成实例时，需要像以前一样使用new关键字，但如果忘记加上new，像函数那样调用Class，将会报错。

```
var point = Point(2, 3);
// 正确
var point = new Point(2, 3);
```

与ES5一样，类的所有实例共享一个原型对象。

```
var p1 = new Point(2,3);
var p2 = new Point(3,2);

p1.__proto__ === p2.__proto__
```

Class不存在变量提升（hoist），这一点与ES5完全不同。

```
new Foo(); // ReferenceError
class Foo {}
```

上面代码中，如果Foo类使用在前，定义在后，这样会报错。因为ES6不会把变量声明提升到代码头部。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

```
{
  let Foo = class {};
  class Bar extends Foo {
  }
}
```

上面的代码不会报错，因为class继承Foo的时候，Foo已经有定义了。但是，如果存在Class的提升，上面代码就会报错，因为class会被提升到代码

头部，而let命令是不提升的，所以导致class继承Foo的时候，Foo还没有定义。

Class之间可以通过extends关键字实现继承，这比ES5通过修改原型链实现继承，要清晰和方便很多。

```
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y); // 调用父类的constructor(x, y)
    this.color = color;
  }

  toString() {
    return this.color + ' ' + super.toString(); // 调用父类的toSt
ring()
  }
}
```

上面代码中，constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的

this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

ES5的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

如果子类没有定义constructor方法，这个方法会被默认添加。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

另一个需要注意的地方是，在子类的构造函数中，只有调用super之后，才可以使用this关键字，否则会报错。这是因为子类实例的构建，是基于对父类实例加工，只有super方法才能返回父类实例。

ES6允许继承原生构造函数定义子类，这意味着，ES6可以自定义原生数据结构（比如Array、String等）的子类，这是ES5无法做到的。下面就是定义了一个带版本功能的数组。

```
class VersionedArray extends Array {
  constructor() {
    super();
    this.history = [[]];
  }
  commit() {
    this.history.push(this.slice());
  }
  revert() {
    this.splice(0, this.length, ...this.history[this.history.le
ngth - 1]);
  }
}

var x = new VersionedArray();

x.push(1);
x.push(2);
x // [1, 2]
x.history // [[]]

x.commit();
x.history // [[], [1, 2]]
x.push(3);
x // [1, 2, 3]

x.revert();
x // [1, 2]
```

与ES5一样，在Class内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

```
class MyClass {
  constructor() {
    // ...
  }
  get prop() {
    return 'getter';
  }
  set prop(value) {
    console.log('setter: '+value);
  }
}

let inst = new MyClass();

inst.prop = 123;
// setter: 123

inst.prop
// 'getter'
```

ES6定义静态方法，也比原来的清晰好多，并且ES6的静态方法是可以被子类继承的。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}
```

```
class Bar extends Foo {
}

Bar.classMethod(); // 'hello'
```

静态方法也是可以从super对象上调用的。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

class Bar extends Foo {
  static classMethod() {
    return super.classMethod() + ', too';
  }
}

Bar.classMethod();
```

最后我们看一下new.target属性。new是从构造函数生成实例的命令。ES6为new命令引入了一个new.target属性，（在构造函数中）返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。

```
function Person(name) {
  if (new.target !== undefined) {
    this.name = name;
  } else {
    throw new Error('必须使用new生成实例');
  }
}

// 另一种写法
function Person(name) {
  if (new.target === Person) {
    this.name = name;
  } else {
    throw new Error('必须使用new生成实例');
  }
}

var person = new Person('张三'); // 正确
var notAPerson = Person.call(person, '张三');  // 报错
```

目前，ES6这个类机制还在不断改良，其最大的践行者React Native甚至提供了一些更好的语法。而无论是ES6的原生语法还是React Native的改良语法，我们无法一时三刻直接在浏览器中直接运行它们。这时建议大家使用babel编译你的JavaScript。比如下面的ES6代码。

```
//定义父类View
class View {
  constructor(options) {
    this.model = options.model;
    this.template = options.template;
```

```
  }

  render() {
    return _.template(this.template, this.model.toObject());
  }
}
//实例化父类View
var view = new View({
  template: 'Hello, <%= name %>'
});
//定义子类LogView，继承父类View
class LogView extends View {
  render() {
    var compiled = super.render();
    console.log(compiled);
  }
}
```

编译后其骨干大概是这样（随着babel的版本，结果当然不同）。借此机会，我们也可以窥探一下ES6的类工厂是怎么样的！

```
var View = (function() {
    function View(options) {
        _classCallCheck(this, View);
            this.model = options.model;
        this.template = options.template;
    }
    _createClass(View, [{
        key: 'render',
        value: function render() {
            return _.template(this.template, this.model.toObjec
t());
        }
    }]);
    return View;
})();
```

```
var LogView = (function(_View) {
    _inherits(LogView, _View);
    function LogView() {
        _classCallCheck(this, LogView);
        _get(Object.getPrototypeOf(LogView.prototype), 'constru
ctor',this).apply(this,arguments);
    }
    _createClass(LogView, [{
        key: 'render',
        value: function render() {
            var compiled = _get(Object.getPrototypeOf(LogView.p
rototype), 'render',
            this).call(this);
            console.log(compiled);
        }
    }]);
    return LogView;

})(View);
```

生成的两个类是包裹在IIFE中，里面是一个同名的函数。这个函数经过`_createClass()`函数的处理之后，被返回了。所以我们得出的第一点结论就是，**ES6** 中的 **class** 实际就是函数 。我们发现，在class中设定的属性被放在ES5的构造函数中，而方法则以键值对的形式传入一个`_createClass()`函数中。那么这个`_createClass()`函数又制造了什么魔法呢？

```
var _createClass = (function() {
    function defineProperties(target, props) {
        for (var i = 0; i < props.length; i++) {
            var descriptor = props[i];
            descriptor.enumerable = descriptor.enumerable || fa
lse;
            descriptor.configurable = true;
            if ('value' in descriptor) descriptor.writable = tr
ue;
            Object.defineProperty(target, descriptor.key, descr
iptor);
        }
    }
    return function(Constructor, protoProps, staticProps) {
        if (protoProps) defineProperties(Constructor.prototype,
 protoProps);
        if (staticProps) defineProperties(Constructor, staticPr
ops);
        return Constructor;
    };
})();
```

`_createClass` 也是一个IIFE，有一个内部的函数defineProperties，这个函数遍历属性的描述符，进行描述符的默认设置，最后使用Object.defineProperty()方法来写入对象的属性。IIFE的renturn部分有两个分支，一个是针对一个类的原型链方法，一个是静态方法，我们看到原型链方法被写入构造函数的原型对象里，而静态方法则被直接写入构造函数里，因此我们不用实例化对象

就可以直接调用一个类的静态方法了。

我们再看ES6中的类继承是如何实现的。

```
function _inherits(subClass, superClass) {
    if (typeof superClass !== 'function' && superClass !== null
) {
        throw new TypeError('Super expression must either be nu
ll or a function, not ' + typeof superClass);
    }
    subClass.prototype = Object.create(superClass && superClass
.prototype, {
        constructor: {
            value: subClass,
            enumerable: false,
            writable: true,
            configurable: true
        }
    });
    if (superClass) Object.setPrototypeOf ? Object.setPrototype
Of(subClass, superClass) : subClass.__proto__ = superClass;
}
```

_inherits()函数的关键部分便是subClass.prototype = Object.create(···)。通过Object.create()方法来指定新创建对象的原型，由此省去了对父类构造函数的处理，达到了简单的原型继承效果。

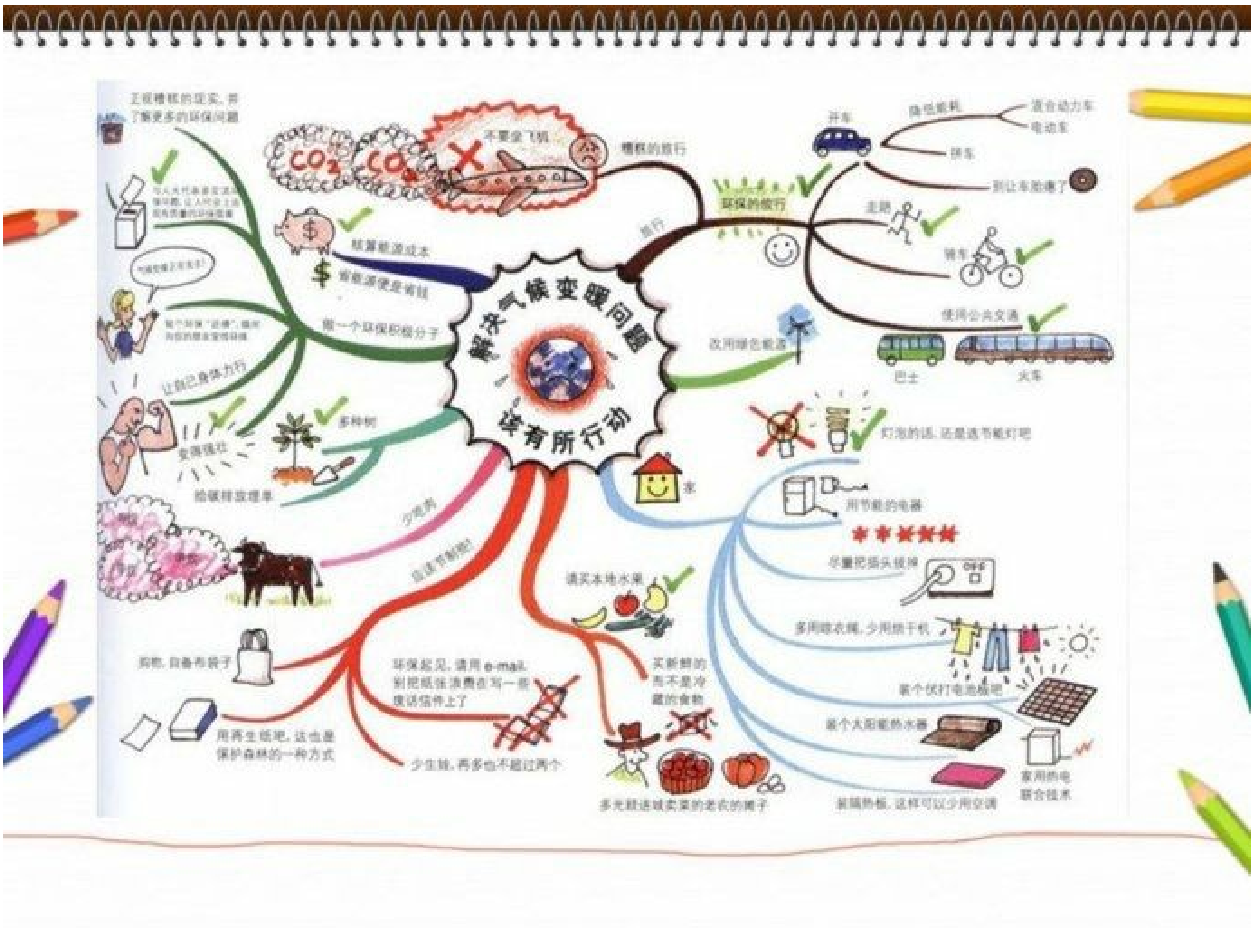
不止如此，子类比起父类也多出了一些东西，最大的不同便是增加了一个`_get()`函数的调用。我们仔细看这个`_get()`函数会发现它接收几个参数，包括子类的原型、**constructor**标识符，还有**this**。再看下面对super.render()的处理，同样是用`_get()`函数来处理的。再看`_get()`函数的源码，就可以发现`_get()`函数的作用便是遍历对象的原型链，找出传入的标识符对应的属性，把它用apply绑定在当前上下文上执行。

最后提一下，虽然ES6类是如此好用方便，但是透过刚才的源码分析，我们知道它是基于Object.defineProperty，而这个API在IE8只是针对节点，因此IE8还是我们一道很难跨过去的槛。惦量一下自家产品的兼容性要求，然后决定要哪种类工厂来组织你的代码吧。

---

[1] Gof, Gang of Four，指称为设计模式先驱的四人：Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides。早在20世纪90年代他们就出版了划时代的著作——《设计模式：可复用面向对象软件的基础》，该书曾被认为是整个软件模式发展的先驱。该书总结的23种设计模式，为我们面向对象开发遇到各种问题提供了思路与模板。

第5章　选择器引擎



jQuery凭借选择器风靡全球，从而使各大框架类库争先开发自己的选择器，一时间选择器成为框架的标配。

其实，早期jQuery选择器与我们现在看到的大不一样。它最初是使用混杂xpath语法的selector，第二代转为纯CSS带自定义伪类（比如从xpath借鉴过来位置伪类）的**Sizzle** 。但Sizzle也一直在变，因为它的关系选择器一直存在问题，因此不断重构，在jQuery 1.9时终于搞定，并最终决定全面支持CSS3的结构伪类。有据可查的早期三大选择器引擎是2003年Simon Willison的**getElementsBySelector** ，然后是2004年Dean Edwards的**cssQuery** ，最后是2005年发布的jQuery。据John Resig在《JavaScript精粹》说，他本来只想写个选择器引擎，但cssQuery“光芒太盛”，无法与之争锋，匆忙间作为一个较为完整的dom类库面世。

2005年，Ben Nolan的Behaviour.js，内置了早以闻名于世的getElementsBySelector，是第一个集成事件处理、CSS风格的选择器引擎与onload处理的类库。

本章介绍如何从头到尾制造一个选择器引擎，在此我们先看看前人的努力吧。

5.1　浏览器内置的寻找元素的方法

请不要追问2005年之前开发人员是怎么在这种“缺东缺西”的环境下干活的，那时浏览器大战打得正酣，程序员们发明了navigator.userAgent检测进行自保！网景战败，因此有关它的记录不多。但IE确确切切留下许多资料，比如取得元素。我们可以直接根据元素的ID就取得元素自身[1]，不通过任何API，自动映射成全局变量。在不关注全局污染时，这是很酷的特性。又如取得所有元素，直接document.all。取得某一种标签类型的元素，只需做一下分类，如P标签、`document.all.tags("p")`。时至今天，IE4里的这个古老API还能在IE10标准模式下正常运作！

有资料可查的是**getElementById**、**getElementsByTagName** 是IE5引入的，那是1999

年的事，与微软另一个辉煌的产品Windows 98捆绑在一起。因此，那时的程序的代码都倾向于为IE做兼容。我在网上找到一个让IE4支持getElementById的代码，刻着时代的“烙印”。

```
var ie4 = document.all && !document.getElementById;
if(ie4) {
  document.getElementById = new Function('var expr = /^\\w[\\w\
\d]*$/,'+
    'elname=arguments[0]; if(!expr.test(elname)) { return null;
 } '+
    'else if(eval("document.all."+elname)) { return '+
    'eval("document.all."+elname); } else return null;')
}
```

此外还有**getElementsByTagName** 的实现如下。

```
function getElementsByTagName(str) {
    if (str == "*") {
        return document.all
    } else {
        return document.all.tags(str)
    }
}
```

但人们很快就发现问题了，IE的

getElementById 是不区分表单元素ID与Name，因此如果有一个表单元素只定义name并与我们的目标元素ID同名，且我们的目标元素在它的后面，那么就会选错元素。这个问题一直延续到IE7。

IE的**getElementsByTagName** 也有问题。当参数为*号通配符时，它会混入注释节点 [2]，并且无法选取Object下的元素。下面是解决办法。

```
//J. Max Wilson
if (/msie/i.test (navigator.userAgent)) {//only override IE
    document.nativeGetElementById = document.getElementById;
    document.getElementById = function(id){
        var elem = document.nativeGetElementById(id);
        if(elem){//IE5
            if(elem.id == id){
                return elem;
            }else{//IE4
                for(var i=1;i<document.all[id].length;i++){
                    if(document.all[id][i].id == id){
                        return document.all[id][i];
                    }
                }
            }
        }
        return null;
    }
}

//Dean Edwards
function getElementsByTagName(node, tagName) {
  var elements = [], i = 0, anyTag = tagName === "*", next = no
de.firstChild;
```

```
  while ((node = next)) {
    if (anyTag ? node.nodeType === 1 : node.nodeName === tagNam
e) elements[i++] = node;
    next = node.firstChild || node.nextSibling;
    while (!next && (node = node.parentNode)) next = node.nextS
ibling;
  }
  return elements;
}
```

此外W3C还提供了一个**getElementsByName** 的方法，这个在IE中使用也有问题，它只能选取表单元素，由于我们后面用不到它，先行略去。

这是Prototype.js到来之前，所有可用的原生选择器。因此Simon Willison搞出getElementsBySelector，让世人眼前一亮。

之后的情况大家应该知道了，出现N个版本的**getElementsBySelector** 。不过大多数是在Simon Willison的基础上改进的，甚至当时还讨论将它标准化！

```
http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2005-Septem
ber/subject.html#4782
```

虽然这个打算最后搁浅了，但Simon Willison的getElementsBySelector代表的是历史的前进方向。jQuery则有点偏向了。Prototype.js则在Ajax热炒浪潮中扶摇直上，Prototype.js 1.4在document添加日后成为标准的**getElementsByClassName** 与失败了的**getElementsBySelector** ，此外还有比此时jQuery更加好用的$$ 。

不过，jQuery最终还是胜利了，Sizzle的设计很特别，各种优化别出心裁。浏览器没有闲着，Netscape借Firefox还魂，挑起第二次浏览器战争，其间往HTML引入XML的xpath，其API为**document.evaluate** 。但xpath又分为level1、level2、level3，各浏览器在不同版本的支持又不一致，加之语法比较复杂，因此普及不开，更甭论存在什么bug。同一时间浏览器还标准化了

getElementsByClassName，但这个API也只有选择器引擎的作者们在类库里面小心翼翼使用，因此它在Safari与Opera存在一些奇怪bug。

微软为了保住占有率，在IE8上加入**querySelector**与**querySeletorAll**，相当于getElementsBySelector的升级版，它还支持前所未有的结构伪类、状态伪类、语言伪类与取反伪类。这时谷歌的Chrome参战，激发标准浏览器的升级热情，IE8新加的选择器大家都支持了，还支持得更标准。此时还出现了一种与选择器功能相反的API——匹配器**matchesSelector**，它对我们编写选择器引擎非常有帮助。现在CSS方面有关selector 4的规范还在起草中，querySeletorAll也暂只支持到selector 3部分，但即便如此，目前带来的兼容性问题已经让选择器引擎作者们为难了！

5.2 getElementsBySelector

我们先来看一下这个最古老的选择器引擎。它规定了今后许多选择器的发展方向。在解读中可能涉及许多概念，但不要紧，后面有更详细的解析。现在只是初步了解一下大概蓝图。

```
/* document.getElementsBySelector(selector)
   New in version 0.4: Support for CSS2 and CSS3 attribute sele
ctors:
   See http://www.w3.org/TR/css3-selectors/#attribute-selectors
   Version 0.4 - Simon Willison, March 25th 2003
   -- Works in Phoenix 0.5, Mozilla 1.3, Opera 7, Internet Expl
orer 6, Internet Explorer 5 on Windows
   -- Opera 7 fails
*/
function getAllChildren(e) {
  //取得一个元素的所有子孙，兼并容IE5
  return e.all ? e.all : e.getElementsByTagName('*');
}

document.getElementsBySelector = function(selector) {
  //如果不支持getElementsByTagName则直接返回空数组
  if (!document.getElementsByTagName) {
    return new Array();
  }
  //切割CSS选择符，分解一个个单元（每个单元可能代表一个或几个选择器，比如p
.aaa则由标签选择器与类选
  //择器组成）
  var tokens = selector.split(' ');
  var currentContext = new Array(document);
  //从左到右检测每个单元，换言之此引擎是自顶向下选元素
  //我们的结果集如果中间为空，那么就立即中止此循环了
  for (var i = 0; i < tokens.length; i++) {
```

```
    //去掉两边的空白（但并不是所有的空白都是没用，
    //两个选择器组之间的空白代表着后代选择器，这要看作者们的各显神通了）
    token = tokens[i].replace(/^\s+/,'').replace(/\s+$/,'');;
    //如果包含ID选择器，这里略显粗糙，因为它可能在引号里面
    //此选择器支持到属性选择器，则代表着它可能是属性值的一部分
    if (token.indexOf('#') > -1) {
      // 这里假设这个选择器组以tag#id或#id的形式组成，可能导致BUG
      //但这暂且不谈，我们还是沿着作者的思路进行下去吧
      var bits = token.split('#');
      var tagName = bits[0];
      var id = bits[1];
      //先用ID值取得元素，然后判定元素的tagName是否等于上面的tagName
      //此处有一个不严谨的地方，element可能为null，会引发异常
      var element = document.getElementById(id);
      if (tagName && element.nodeName.toLowerCase() != tagName)
 {
        // 没有直接返回空结果集
        return new Array();
      }
      //置换currentContext，跳至下一个选择器组
      currentContext = new Array(element);
      continue;
    }
    // 如果包含类选择器，这里也假设它以.class或tag.class的形式
    if (token.indexOf('.') > -1) {

      var bits = token.split('.');
      var tagName = bits[0];
      var className = bits[1];
      if (!tagName) {
        tagName = '*';
      }
      // 从多个父节点出发，取得它们的所有子孙，
      // 这里的父节点即包含在currentContext的元素节点或文档对象
      var found = new Array;//这里是过滤集，通过检测它们的className
决定去留
      var foundCount = 0;
      for (var h = 0; h < currentContext.length; h++) {
        var elements;
        if (tagName == '*') {
           elements = getAllChildren(currentContext[h]);
        } else {
           elements = currentContext[h].getElementsByTagName(ta
gName);
        }
```

```
        for (var j = 0; j < elements.length; j++) {
          found[foundCount++] = elements[j];
        }
      }
      currentContext = new Array;
      var currentContextIndex = 0;
      for (var k = 0; k < found.length; k++) {
        //found[k].className可能为空，因此不失为一种优化手段，但new R
egExp放在//外围更适合
        if (found[k].className && found[k].className.match(new
RegExp('\\b'+className+
        '\\ b'))) {
          currentContext[currentContextIndex++] = found[k];
        }
      }
      continue;
    }
    //如果是以tag[attr(~|^$*)=val]或[attr(~|^$*)=val]的形式组合
    if (token.match(/^(\w*)\[(\w+)([=~\|\^\$\*]?)=?"?([^\]")*)"
?\)$/)) {
      var tagName = RegExp.$1;
      var attrName = RegExp.$2;
      var attrOperator = RegExp.$3;
      var attrValue = RegExp.$4;
      if (!tagName) {
        tagName = '*';
      }
      // 这里的逻辑以上面的class部分相似，其实应该抽取成一个独立的函数
      var found = new Array;
      var foundCount = 0;
      for (var h = 0; h < currentContext.length; h++) {
        var elements;
        if (tagName == '*') {
            elements = getAllChildren(currentContext[h]);
        } else {
            elements = currentContext[h].getElementsByTagName(t
agName);
        }
        for (var j = 0; j < elements.length; j++) {
          found[foundCount++] = elements[j];
        }
      }
      currentContext = new Array;
      var currentContextIndex = 0;
      var checkFunction;
```

```
      //根据第二个操作符生成检测函数，后面章节会详解，这里不展开
      switch (attrOperator) {
        case '=': //
          checkFunction = function(e) { return (e.getAttribute(
attrName) == attrValue); };
          break;
        case '~':
          checkFunction = function(e) { return (e.getAttribute(
attrName).match(new RegExp
          ('\\b'+attrValue+'\\b'))); };
          break;
        case '|':
          checkFunction = function(e) { return (e.getAttribute(
attrName).match(new RegExp
          ('^'+attrValue+'-?'))); };
          break;
        case '^':
          checkFunction = function(e) { return (e.getAttribute(
attrName).indexOf
          (attrValue) == 0); };
          break;
        case '$':
          checkFunction = function(e) { return (e.getAttribute(
attrName).lastIndexOf
          (attrValue) == e.getAttribute(attrName).length - attr
Value.length); };
          break;
        case '*':
          checkFunction = function(e) { return (e.getAttribute(
attrName).indexOf(attrValue)
          > -1); };
          break;
        default :
          checkFunction = function(e) { return e.getAttribute(a
ttrName); };
      }
      currentContext = new Array;
      var currentContextIndex = 0;
      for (var k = 0; k < found.length; k++) {
        if (checkFunction(found[k])) {
          currentContext[currentContextIndex++] = found[k];
        }
      }
      continue;
    }
```

```
    // 如果没有"#"，"."，"["这样的特殊字符，我们就当成是tagName
    tagName = token;
    var found = new Array;
    var foundCount = 0;
    for (var h = 0; h < currentContext.length; h++) {
      var elements = currentContext[h].getElementsByTagName(tag
Name);
      for (var j = 0; j < elements.length; j++) {
        found[foundCount++] = elements[j];
      }
    }
    currentContext = found;
  }
  return currentContext;//最后返回结果集
}
```

显然受当时的网速限制，页面不会很大，也不可能发展起复杂的交互，因此JavaScript还没有到大规模使用的阶段，我们看到那时的库也不怎么重视全局污染。主要API直接在document上操作，参数只有一个CSS表达符。从我们的分析来看，它并不支持联选择器（后面介绍），并且要求每个选择器组不能超出两个，否则报错。换言之，它只对下面这样形式的CSS表达式有效。

```
#aa p.bbb [ccc=ddd]
```

CSS表达符将以空白分割成多个选择器组，每个选择器不能超过两种选择器类型，并且其中一种为标签选择器。

作为早期的选择器，它也没有像以后那样对结果集进行去重，把元素逐个按照文档出现的顺序进行排序。我们在第一节指出的bug，它也没有规避，这可能受当时JavaScript技术交流太少所致。这些都是我们是日后要改进的地方。

5.3 选择器引擎涉及的知识点

这一节我们开始学习一下5.2节中介绍的大量概念。其中，有关选择器引擎实现的概念大多数是笔者从Sizzle中抽取出来的，而CSS表达符部分则是W3C提供的。那么我们先从**CSS** 表达符 部分讲起吧。

选择符 是指一条 **CSS** 样式规则 的最左边的部分，如图5-1所示。

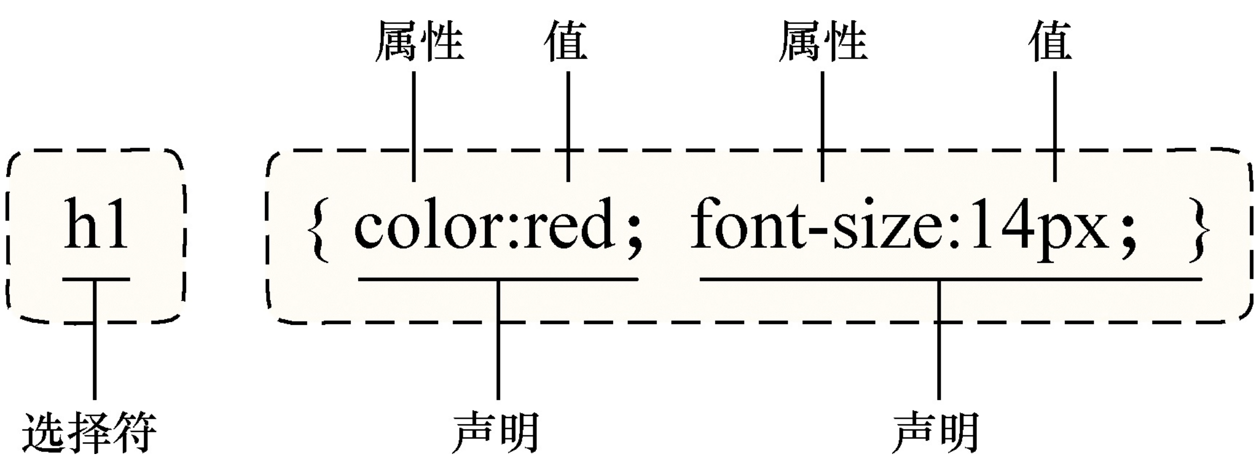


图5-1

上面的只是理想情况，重构人员交给我们的CSS文件，里面的选择符可是复杂多了。选择符混杂着大量的标记，可以分割为许多更细的单元。总的来说，分为**4** 大类 **16** 种 。此外，还没有包含选择器引擎无法操作的伪元素。

4大类是指并联选择器 、简单选择器 、关系选择器 与伪类 。

并联选择器 有1种：逗号“ ,”，一种不是选择器的选择器，用于合并多个分组的结果。

简单选择器 分5种：**ID** 、标签 、类 、属性 、通配符 。

关系选择器 分4种：亲子 、后代 、相邻 、兄长 。

伪类 分6种：动作伪类 、目标伪类 、语言伪类 、状态伪类 、结构伪类 、取反伪类 。

jQuery还添加了3种伪类：可见性伪类 :visible、内容伪类 :content()、包含伪类 :has()。

简单选择器 又称为基本选择器 ，这是在Prototype.js之前的选择器都已经支持的选择器类型。不过CSS上，IE7才支持部分属性选择器。其中，它们设计得非常整齐划一，我们可以通过它的第一个字符决定它们的类型。比如：ID选择器的第一个字符为**#** ，类选择器为**.** ，属性选择器为**[** ，通配符选择器为***** ；标签选择器为不包含特殊字符的英文数字组合 ，jQuery就是使用`isTag = !/\W/.test(part)`进行判定的。

在实现上，我们在这里有许多原生API可用，如getElementById、getElementsByTagName、getElementsByClassName、document.all。属性选择器可以用getAttribute、getAttributeNode、attributes、hasAttribute，2003年曾讨论引入

getElementsByAttribute，但没有成功，Firefix上XUI的同名API就是当时的产物。不过属性选择器的确比较复杂，历史上它是分两步实现的。

CSS2.1中，属性选择器有以下4种形态。

```
+ [att]：选取设置了att属性的元素，不管设定的值是什么。
+ [att=val]：选取所有att属性的值完全等于 val 的元素。
+ [att~=val]：表示一个元素拥有属性 att，值为一个被空格隔开的多个字符串，只要其中一个字符串等于val就能匹配上。基于此特性，如果浏览器不支持geElementsByClassName，我们可以将.aaa转换为[class~=aaa]来处理。
+ [att|=val]：表示一个元素拥有属性 att，并且该属性含 'val' 或以 'val-' 开头。
```

CSS3中，属性选择器又增加3种形态。

```
+ [att^=val]：选取所有att属性的值以val开头的元素。
+ [att$=val]：选取所有att属性的值以val结尾的元素。
+ [att*=val]：选择所有att属性的值包含val字样的元素。以上3者我们都可以通过indexOf轻松实现。
```

此外，大多数选择器引擎，还实现了一种[att!=val]的自定义属性选择器。意思很简单，选取所有att属性不等于val的元素，这正好与[att=val]相

反。这个我们可以通过CSS3的取反伪类实现。

5.3.1　关系选择器

关系选择器是不能单独存在的，它必须夹在其他种类的选择器中使用，但某些选择器引擎可能允许它放在最开始的位置。在很长时间内，只存在后代选择器（**E　F**），就是两个简单选择器E与F之间的空白。CSS2.1又添加了两个，亲子选择器（**E > F**）与相邻选择器（**E + F**），它们也夹在两个简单选择器之间，但允许大于号或加号两边存在空白，这时，空白就不是表示后代选择器。CSS3又添加了一个，兄长选择器（**E～F**），规则同上。CSS4又增加了一个父亲选择器[3]，不过其规则一直在变，这里就不说了。

1．后代选择器

通常我们在引擎内构建一个getAll的函数，要求传入一个文档对象或元素节点取得其子孙。这里

要特别注意IE下**document.all**、**getElementsByTagName("*")** 混入注释节点的问题。

2．亲子选择器

这个我们如果不打算兼容XML，那么直接使用children就行了。不过IE5～IE8它都会混入注释节点，表5-1是兼容列表。

表**5-1**

Chrome	Firefox	IE	Opera	Safari
1+	3.5+	5+	10+	4+

```
function getChildren(el) {
    if (el.childElementCount) {
        return [].slice.call(el.children);
    }
    var ret = [];
    for (var node = el.firstChild; node; node = node.nextSiblin
g) {
        node.nodeType == 1 && ret.push(node);
    }
    return ret;
}
```

3. 相邻选择器

相邻选择器就是取得当前元素向右的一个元素节点，视情况使用nextSibling或nextElementSibling。

```
function getNext(el) {
    if ("nextElementSibling" in el) {
        return el.nextElementSibling
    }
    while (el = el.nextSibling) {
        if (el.nodeType === 1) {
           return el
        }
    }
    return null;
}
```

4. 兄长选择器

兄长选择器就是取其左边的所有同级元素节点。

```
function getPrev(el) {
    if ("previousElementSibling" in el) {
        return el.previousElementSibling;
    }
    while (el = el.previousSibling) {
```

```
        if (el.nodeType === 1) {
            return el;
        }
    }
    return null;
}
```

上面提到的childElementCount和nextElementSibling都是2008年12月通过Element Traversal规范的，用于遍历元素节点。加上后来补充的parentElement，我们查找元素就非常方便，如表5-2所示。

表**5-2**

	遍历所有子节点	遍历所有子元素
最前的	firstChild	firstElementChild
最后的	lastChild	lastElementChild
前面的	previousSibling	previousElementSibling
后面的	nextSibling	nextElementSibling
上面的	parentNode	parentElement

长度	length	childElementCount

5.3.2　伪类

伪类 是选择器中最庞大的家族，从CSS1开始支持，以字符串开头。在CSS3，出现了要求传参的结构伪类与取反伪类。

1．动作伪类

动作伪类又分为链接伪类和用户行为伪类，其中链接伪类由:visited和:link组成，用户行为伪类由:hover、:active和:focus组成。这里我们基本上只能模拟:link，而在浏览器原生的querySelectorAll对它们的支持也存在差异，IE8～IE10取:link存在错误，它只能取A标签，实际:link是指代A、AREA、LINK这3种标签，而其他标签浏览器都正确。另外，除Opera，Safari外，其他浏览器取:focus都正常。除Opera外，其他浏览器取:hover都正确。剩下

的:active与:visited都为零。下面是测试页面。

```
<!DOCTYPE HTML>
<html>
    <head>
        <title></title>
        <link href="aa" type="text/css" rel="stylesheet" charse
t="utf-8" />
        <meta http-equiv="Content-Type" content="text/html; cha
rset=UTF-8">
        <script>
            window.onload = function() {
                document.querySelector("#aaa").onclick = functi
on() {
                    alert(document.querySelectorAll(":focus").l
ength);//1
                }
                document.querySelector("#bbb").onmouseover = fu
nction() {
                    //4 html, body, p, a
                    alert(document.querySelectorAll(":hover").l
ength);
                }
            }

            function test() {
                alert(document.querySelectorAll(":link").length
);// 6
            }
        </script>
    </head>
    <body>
        <p><a href="javascript:void 0" id="aaa">aaa</a></p>
        <p><a href="javascript:void 0" id="bbb">bbb</a></p>
        <button type="button" onclick="test()">点我</button>
        <img src="planets.jpg" border="0" usemap="#planetmap" a
lt="Planets" />
        <map name="planetmap" id="planetmap">
           <area shape="circle" coords="180,139,14" href ="venu
s.html" alt="Venus" />
           <area shape="circle" coords="129,161,10" href ="merc
ur.html" alt="Mercury" />
           <area shape="rect" coords="0,0,110,260" href ="sun.h
```

```
tml" alt="Sun" />
        </map>
    </body>
</html>
```

伪类没有专门的API得到结果集，因此我们需要通过上一次得到的结果集进行过滤。在浏览器中，我们可以通过document.links得到部分结果，因为它不包含LINK标签。因此，最好的方法是判定它的tagName是否等于**A**、**AREA**、**LINK**的其中一个。

2. 目标伪类

目标伪类即:target伪类，指其id或者name属性与URL中的hash部分（即#之后的部分）匹配上的元素。

譬如文档中有一个元素，其id为`section_2`，而URL中的hash部分也是`#section_2`，那么它就是我们要取的元素。

Sizzle中的过滤函数如下。

```
"target": function(elem) {
    var hash = window.location && window.location.hash;
    return hash && hash.slice(1) === elem.id;
},
```

3. 语言伪类

语言伪类即:lang伪类，用来设置使用特殊语言内容的样式。比如:lang(de)的内部应该为德语，需要特殊处理。

注意lang虽然作为DOM元素的一个属性，但:lang伪类与属性选择器有所不同，具体表现在:lang伪类具有“继承性”，如下面HTML表示的文档。

```
<body lang="de"><p>一个段落</p></body>
```

如果使用[lang=de]则只能选择到body元素，因

为 **p** 元素没有 **lang** 属性 。但是使用:lang(de)则可以同时选择到body和p元素，因为依类具有继承的特性 。

Sizzle中的过滤函数如下。

```
//https://github.com/jquery/sizzle/blob/2.3.0/src/sizzle.js#L14
18-L1437
"lang": markFunction( function( lang ) {
    // lang value must be a valid identifier
    if ( !ridentifier.test(lang || "") ) {
        Sizzle.error( "unsupported lang: " + lang );
    }
    lang = lang.replace( runescape, funescape ).toLowerCase();
    return function( elem ) {
        var elemLang;
        do {
            if ( (elemLang = documentIsHTML ?
                elem.lang :
                elem.getAttribute("xml:lang") || elem.getAttrib
ute("lang")) ) {

                elemLang = elemLang.toLowerCase();
                return elemLang === lang || elemLang.indexOf( l
ang + "-" ) === 0;
            }
        } while ( (elem = elem.parentNode) && elem.nodeType ===
 1 );
        return false;
    };
}),
```

对比mass的实现如下。

```
lang: { //标准 CSS3语言伪类
    exec: function(flags, elems, arg) {
        var result = [],
            reg = new RegExp("^" + arg, "i"),
            flag_not = flags.not;
        for (var i = 0, ri = 0, elem; elem = elems[i++];) {
            var tmp = elem;
            while (tmp && !tmp.getAttribute("lang"))
            tmp = tmp.parentNode;
            tmp = !! (tmp && reg.test(tmp.getAttribute("lang"))
);
            if (tmp ^ flag_not) result[ri++] = elem;
        }
        return result;
    }
},
```

4．状态伪类

状态伪类用于标记一个UI元素的当前状态，由:checked、:enabled、:disabled和:indeterminate这4个伪类组成。我们可以分别通过元素的checked、disabled、indeterminate属性进行判定。

5．取反伪类

取反伪类即:not伪类，其参数为一个或多简单选择器，里面用逗号隔开。在jQuery等选择器引擎

中允许你传入其他类型的选择器，甚至可以进行多个取反伪类套嵌。

5.3.3　其他概念

种子集：或者叫候选集。如果CSS选择符非常复杂，我们要分几步才能得到我们想要的元素，那么第一次得到的元素集合就叫种子集。在Sizzle这样基本从右到左，它的种子集中就有一部分为我们最后得到的元素。如果选择器引擎是从左到右选择器，那么它们只是我们继续查它们的孩子或兄弟的“据点”而已。

结果集：选择器引擎最终返回的元素集合，现在约定俗成，它要保持与querySelectorAll得到的结果一致，即要求没有重复元素，且顺序与它们在DOM树上出现的顺序一致。

过滤集：我们选取一组元素后，它之后每一

个步骤要处理的元素集合都可以称之为过滤集。比如p.aaa，如果浏览器不支持querySelectorAll，但支持getElementsByClassName，那么我们就用它得到种子集，然后在循环中通过tagName==="P"进行过滤。若不支持，只能通过getElementsByTagName得到种子集，然后通过className进行过滤。显然大多数情况下，前者比后者快多了。同理，如果它们之间存在ID选择器，由于ID在一个文档中不允许重复，因此使用ID进行查找更快。在Sizzle中，如果不支持querySelectorAll，它会智能地以ID、Class、Tag的顺序进行查找。

选择器群组：一个选择符被并联选择器划分成的每一个大分组。

选择器组：一个选择器群组被关系选择器划分的第一个小分组。考虑到性能，每一个小分组建议都加上tagName，因为这样在IE6会方便我们使用

getElementsByTagName。比如div p.aaa比div .aaa快多了。前者是通过两次getElementsByTagName寻找，最后用className过滤，后者是通过getElementsByTagName得到种子集，然后再取它们的所有子孙，明显这样得到的过滤集比前者的数量多很多。从实现上说，你可以选择从左到右，也可以像Sizzle那样从右到左，但Sizzle只能说大体上是这个方向，实际情况复杂多了。

另外，选择器也分为编译型与非编译型。编译型是EXT发明的，这个阵营的选择器中有EXT、QWrap、NWMatchers、JindoJS。非编译型的就更多了，如Sizzle、Icarus（mass Framework的选择器引擎）、Slick（mootools的选择器）、YUI、dojo、uupaa、peppy……

还有一种利用xpath实现的选择器，最著名的是Base2。它先实现了xpath那一套，方便IE也能使用

document. evaluate，然后将CSS选择符翻译成xpath。其他比较出名的有casperjs、DOMAssistant等。

像Sizzle、mootools、Icarus等还支持选择XML元素，因为XML是一种重要的数据传输格式，后端通过XHR返回的可能就是XML，这样我们通过选择器引擎抽取所需要的数据就简单多了。

5.4　选择器引擎涉及的通用函数

5.4.1　isXML

最强大的前几名选择器引擎都能操作XML文档，但XML与HTML存在很大的差异，没有className和getElementById，并且nodeName是区分大小写的，在旧版本IE中还不能直接给XML元素添加自定义属性。因此区分XML和HTML是非常有必要的，我们看一下各大引擎的实现吧。

Sizzle的实现如下：

```
var isXML = Sizzle.isXML = function( elem ) {
    var documentElement = elem && (elem.ownerDocument || elem).
documentElement;
    return documentElement ? documentElement.nodeName !== "HTML
" : false;
}
```

但这样不严谨，因为XML的根节点可能是

HTML标签，比如这样创建一个XML文档：

```
try {
    var doc = document.implementation.createDocument(null, 'HTM
L', null);
    alert(doc.documentElement)
    alert(isXML(doc))
} catch (e) {
    alert("不支持creatDocument方法")
}
```

我们来看mootools的slick的实现。

```
var isXML = function(document) {
    return ( !! document.xmlVersion) || ( !! document.xml) || (
toString.call(document) == '[object XMLDocument]') || (document
.nodeType == 9 && document.documentElement.nodeName != 'HTML');
};
```

mootools用到了大量属性来进行判定，从mootools1.2到现在还没什么改动，说明还是很可靠的。我们再精简一下。在标准浏览器中，暴露了一个创建HTML文档的构造器HTMLDocument，而IE下的XML元素又拥有selectNodes。

```
var isXML = window.HTMLDocument ? function(doc) {
```

```
    return !(doc instanceof HTMLDocument)
} : function(doc) {
    return "selectNodes" in doc
}
```

不过这些方法都只是规范，JavaScript对象可以随意添加，属性法很容易被攻破，最好是使用功能法。无论XML或HTML文档都支持createElement方法，我们判定创建了元素的nodeName是否区分大小写。

```
var isXML = function(doc) {
    return doc.createElement("p").nodeName !== doc.createElemen
t("P").nodeName;
};
```

这是我当前能给出的最严谨的函数了。

5.4.2　contains

contains方法就是判定参数1是否包含参数2。这通常用于优化，比如早期的Sizzle，对于#aaa p.class这个选择符，它会优先用

getElementsByClassName或getElementsByTagName取种子集，然后就不继续往左走了，直接跑到最左的#aaa，取得#aaa元素，然后通过contains方法进行过滤。随着Sizzle的体积进行增大，它现在只剩下另一个关于ID的用法，即，如果上下文对象非文档对象，那么它会取得其ownerDocument，这样就可以用getElementById，然后利用contains方法进行验证！

```
//Sizzle 1.10.15
var newContext = context.nodeType === 9 ? context : context.own
erDocument
if ( newContext &&
        (elem = newContext.getElementById( m )) &&
        contains( context, elem ) &&
        elem.id === m ) {
        results.push( elem );
        return results;
}
```

我们再看看contains的实现。

```
//Sizzle 1.10.15
var rnative = /^[^{]+\{\s*\[native \w/,
hasCompare = rnative.test( docElem.compareDocumentPosition ),
contains = hasCompare || rnative.test(docElem.contains) ?
        function(a, b) {
```

```
            var adown = a.nodeType === 9 ? a.documentElement :
a,
                    bup = b && b.parentNode;
            return a === bup || !!(bup && bup.nodeType === 1 &&
 (
                    adown.contains ?
                    adown.contains(bup) :
                    a.compareDocumentPosition && a.compareDocum
entPosition(bup) & 16
                    ));
        } :
        function(a, b) {
            if (b) {
                while ((b = b.parentNode)) {
                    if (b === a) {
                        return true;
                    }
                }
            }
            return false;
        };
```

它自己做了预判定，但这时如果传入XML元素节点，可能就会出错。因此建议改成实时判定，虽然每次进入都需要判定一次使用哪个原生API。

现在来解释一下**contains**与**compareDocumentPosition** 这两个API。contains原来是IE的私有实现，后来其他浏览器也借鉴这方法，如Firefox在9.0中也装了此方法。它是一个元素

节点的方法，如果另一个等于或包含于它的内部，就返回true。

compareDocumentPosition是DOM Level 3 specification定义的方法，Firefox等标准浏览器都支持，它用于判定两个节点的关系，而不单止是包含关系。这里是从NodeA.compareDocumentPosition(NodeB)返回的结果，包含你可以得到的信息，如表5-3所示。

表**5-3**

Bits	**interger**	说明
000000	0	元素一致
000001	1	节点在不同的文档（或者一个在文档之外）
000010	2	节点B在节点A之前
000100	4	节点A在节点B之前
001000	8	节点B包含节点A

010000	16	节点A包含节点B
100000	32	浏览器的私有使用

有时候，两个元素的位置关系可能连续满足上表的两种情况，比如A包含B，并且A在B的前面，那么compareDocumentPosition就返回20。

由于旧版本IE不支持compareDocumentPosition，因此jQuery的作者John Resig写了个兼容函数**sourceIndex**，用到IE的另一个私有实现。**sourceIndex** 会根据元素的位置从上到下，从左到右依次加1，比如HTML标签的sourceIndex为0，HEAD标签的为1，BODY标签为2，HEAD的第一个子元素为3……如果元素不在DOM树，那么返回−1。

```
// Compare Position - MIT Licensed, John Resig
function comparePosition(a, b) {
    return a.compareDocumentPosition ? a.compareDocumentPositio
n(b) :
     a.contains ? (a != b && a.contains(b) && 16)+
      (a != b && b.contains(a) && 8)+
```

```
        (a.sourceIndex >= 0 && b.sourceIndex >= 0 ?
            (a.sourceIndex < b.sourceIndex && 4)+
            (a.sourceIndex > b.sourceIndex && 2): 1): 0;
}
```

5.4.3　节点排序与去重

为了让选择器引擎搜到的结果集尽可能接近原生API的结果（因为在最新的浏览器中，我们可能只使用querySelectorAll实现），我们需要让元素节点按它们在DOM树出现的顺序排序。

在IE及Opera早期的版本，我们可以使用**sourceIndex** 进行排序。

标准浏览器可以使用**compareDocumentPosition**，上面不是介绍它可以判定两个节点的位置关系吗？我们只要将它们的结果按位与4不等于0就知道其前后顺序了。

此外，标准浏览器的Range对象有一

个**compareBoundaryPoints** 方法，它也能迅速得到两个元素的前后顺序。

```
var compare = comparerange.compareBoundaryPoints(how, sourceRan
ge);
```

compare：其值可能为1、0、−1（0为相等；1为comparerange在sourceRange之后；−1为comparerange在sourceRange之前）。

how：比较哪些边界点，为常数。

```
（1）Range.START_TO_START——比较两个Range节点的开始点。
（2）Range.END_TO_END——比较两个Range节点的结束点。
（3）Range.START_TO_END——用sourceRange的开始点与当前范围的结束点比较。
（4）Range.END_TO_START——用sourceRange的结束点与当前范围的开始点比较。
```

特别的情况发生于要兼容旧版本标准浏览器与XML文档时，这时只有一些很基础的DOM API，我们需要使用nextSibling来判定谁是哥哥，谁是弟弟。如果它们不是同一个父节点，我们就需要将问

题转化为求最近公共祖先，判定谁是父亲，谁是伯父。到这里，已经是纯算法的问题了。实现的思路有许多，最直观也最笨的做法是，不断向上获取它们的父节点，直到HTML元素连同最初的那个节点组成两个数组，然后每次取数组最后的元素进行比较。如果相同就去掉，一直去掉到不相同为止，最后用nextSibling结束。下面是测试代码，自己找一个HTML页面做标本。

```
window.onload = function() {
    function shuffle(a) {
        //洗牌
        var array = a.concat();
        var i = array.length;
        while (i) {
            var j = Math.floor(Math.random() * i);
            var t = array[--i];
            array[i] = array[j];
            array[j] = t;
        }
        return array;
    }
    var log = function(s) {
        //查看调试消息
        window.console && window.console.log(s)
    }
    var sliceNodes = function(arr) {
        //将NodeList转换为纯数组
        var ret = [],
                i = arr.length;
        while (i)
           ret[--i] = arr[i];
```

```
        return ret;
    }

    var sortNodes = function(a, b) {
        //节点排序
        var p = "parentNode",
                ap = a[p],
                bp = b[p];
        if (a === b) {
            return 0
        } else if (ap === bp) {
            while (a = a.nextSibling) {
                if (a === b) {
                    return -1
                }
            }
            return 1
        } else if (!ap) {
            return -1
        } else if (!bp) {
            return 1
        }
        var al = [],
                ap = a
        //不断往上取，一直取到HTML
        while (ap && ap.nodeType === 1) {
            al[al.length] = ap
            ap = ap[p]
        }
        var bl = [],
                bp = b;
        while (bp && bp.nodeType === 1) {
            bl[bl.length] = bp
            bp = bp[p]
        }
        //然后逐一去掉公共祖先
        ap = al.pop();
        bp = bl.pop();
        while (ap === bp) {
            ap = al.pop();
            bp = bl.pop();
        }
        if (ap && bp) {
            while (ap = ap.nextSibling) {
                if (ap === bp) {
```

```
                    return -1
                }
            }
            return 1
        }
        return ap ? 1 : -1
    }

    var els = document.getElementsByTagName("*")
    els = sliceNodes(els);         //转换成纯数组
    log(els);
    els = shuffle(els); //洗牌（模拟选择器引擎最初得到的结果集的情况）
    log(els);
    els = els.sort(sortNodes); //进行节点排序
    log(els);
}
```

此外，我们还有一种方法，就是选择结束后，用document.getElementsByTagName("*")得到所有元素节点，这时它们肯定是排好序的。我们再依次为它们添加一个类似sourceIndex的自定义属性，值为它的索引值，接下来怎么做就无需我多言了。

好了，我们暂且放下，看一下各大浏览器的实现。

Mootools的Slick引擎，它的比较函数已经注明

是来自Sizzle的（准确来说Sizzle1.6左右，现在它也被改得面目全非）。

```
features.documentSorter = (root.compareDocumentPosition) ? func
tion(a, b) {
    if (!a.compareDocumentPosition || !b.compareDocumentPositio
n)
        return 0;
    return a.compareDocumentPosition(b) & 4 ? -1 : a === b ? 0
: 1;
} : ('sourceIndex' in root) ? function(a, b) {
    if (!a.sourceIndex || !b.sourceIndex)
        return 0;
    return a.sourceIndex - b.sourceIndex;
} : (document.createRange) ? function(a, b) {
    if (!a.ownerDocument || !b.ownerDocument)
        return 0;
    var aRange = a.ownerDocument.createRange(),
            bRange = b.ownerDocument.createRange();
    aRange.setStart(a, 0);
    aRange.setEnd(a, 0);
    bRange.setStart(b, 0);
    bRange.setEnd(b, 0);
    return aRange.compareBoundaryPoints(Range.START_TO_END, bRa
nge);
} : null;
```

它没有打算支持XML与旧版本标准浏览器，不支持就不排序。

mass Framework的Icarus引擎，它结合了一位编程高手JK给出的算法，在排序去重上远胜

Sizzle。突破点在于，无论Sizzle或者Slick，它们都是通过传入比较函数进行排序。而数组的原生sort方法[4]，当它传一个比较函数时，不管它内部用哪种排序算法，都需要多次比对，所以非常耗时。如果能设计让排序在不传参的情况进行，那么速度就会提高！

下面是具体思路（当然只能用于IE或早期Opera）。

（1）取出元素节点的sourceIndex值，转换成一个String对象。

（2）将元素节点附在String对象上。

（3）用String对象组成数组。

（4）用原生的sort进String对象数组排序。

（5）在排好序的String数组中，按序取出元素

节点，即可得到排好序的结果集。

```
function unique(nodes) {
    if (nodes.length < 2) {
        return nodes;
    }
    var result = [],
        array = [],
        uniqResult = {},
        node = nodes[0],
        index, ri = 0,
        sourceIndex = typeof node.sourceIndex === "number",
        compare = typeof node.compareDocumentPosition == "funct
ion";
//如果支持sourceIndex，我们将使用更为高效的节点排序
//http://www.cnblogs.com/jkisjk/archive/2011/01/28/array_quickl
y_sortby.html
    if (!sourceIndex && !compare) { //用于旧版本IE的XML
        var all = (node.ownerDocument || node).geElementsByTagN
ame("*");
        for (var index = 0; node = all[index]; index++) {
            node.setAttribute("sourceIndex", index);
        }
        sourceIndex = true;
    }
    if (sourceIndex) { //IE opera
        for (var i = 0, n = nodes.length; i < n; i++) {
            node = nodes[i];
            index = (node.sourceIndex || node.getAttribute("sou
rceIndex")) + 1e8;
            if (!uniqResult[index]) {        //去重
                (array[ri++] = new String(index))._ = node;
                uniqResult[index] = 1;
            }
        }
        array.sort();                        //排序
        while (ri)
            result[--ri] = array[ri]._;
        return result;
    } else {
        nodes.sort(sortOrder);                //排序
        if (sortOrder.hasDuplicate) {        //去重
            for (i = 1; i < nodes.length; i++) {
```

```
                if (nodes[i] === nodes[i - 1]) {
                    nodes.splice(i--, 1);
                }
            }
        }
        sortOrder.hasDuplicate = false;    //还原
        return nodes;
    }
}

function sortOrder(a, b) {
    if (a === b) {
        sortOrder.hasDuplicate = true;
        return 0;
    } //现在标准浏览器的HTML与XML好像都支持compareDocumentPosition
    if (!a.compareDocumentPosition || !b.compareDocumentPositio
n) {
        return a.compareDocumentPosition ? -1 : 1;
    }
    return a.compareDocumentPosition(b) & 4 ? -1 : 1;
}
```

5.4.4 切割器

选择器降低了JavaScript的入行门槛，它们在选择元素时都很随意，随心所欲，一级级地往上加ID类名，导致选择符非常长。因此如果不支持querySelectorAll，没有一个原生API能承担这工作。因此我们通常使用正常用户的选择符进行切割。这个步骤有点像编译原理的词法分析，拆分出

有用的符号法出来。

这里就拿Icarus的切割器来举例，看它是怎么一步步进化，就知道这工作需要多少细致，如图5-2所示。



图5-2

比如，对于`.td1,div a,body`，上面的正则可以完美将它分解为如下数组。

```
[".td1",",","div"," ", "*", ",", "body"]
```

然后我们就可以根据这个符号流进行工作。由于没有指定上下文对象，就从document开始，发现第一个是类选择器，可以用**getElementsByClassName**，如果没有原生的，

我们仿造一个也不是难事。然后是并联选择器，将上面得到的元素放进结果集。接着是标签选择器，使用**getElementsByTagName**。接着发现是后代选择器，这里可以优化，我们可以预先查看下一个选择器群组是什么，发现是通配符选择器，因此继续使用**getElementsByTagName**。接着又是并联选择器，将上面结果放入结果集。最后一个是标签选择器，又使用**getElementsByTagName**。最后是去重排序。

显然有了切割好的符号，工作简单多了。

但没有东西一开始就是完美的，比如我们遇到这样一个选择符：`:nth-child(2n+1)`。这是一个单独的子元素过滤伪类，它不应该在这里被分析。后面有专门的正则对它的伪类名与传参进行处理。在切割器里，它能得到的最小词素是选择器！

于是切割器改进如下：

```
//让小括号里面的东西不被切割
var reg = /[\w\u00al-\uFFFF][\w\u00al-\uFFFF-]*|[#.:\[](?:[\w\u
00al-
\uFFFF-]|\([^\)]*\)|\])+1(?:\s*)[>+-,*](?:\s*)|\s+/g
```

我们不断增加测试样例，就会发现越来越多问题。又如这样一个选择符：.td1[aa='>111']，属性选择器被拆碎了！

```
[".td1","[aa",">","111"]
```

正则改进如下：

```
//确保属性选择器作为一个完整的词素
var reg = /[\w\u00al-\uFFFF][\w\u00al-\uFFFF-]*|[#.:](?:[\w\u00
al-\uFFFF-
]|\S*\([^\)]*\)+|\[[^\]]*\]|(?:a*)[>+-,*](?:\s*)|\s+/g
```

对于td + div span ，如果最后有一大堆空白，会导致解析错误，我们确保后代选择器夹在两个选择器之间。

```
["td", "+", "div"," ","span", " "]
```

最后一个选择器会被我们的引擎认作是后代选择器，需要提前去掉。

```
//缩小后代选择器的范围
var reg = /[\w\u00al-\uFFFF][\w\u00al-\uFFFF-]*|[#.:](?:[\w\u00
al-\uFFFF-
]|\S+\([^\)]*\))+|\[[^\]]*\]|(?:a*)[>+-,*](?:\s*)|\s(?=[\w\u00a
l-
\uFFFF*#.[:1]/g
```

如果我们也想把最前面的空白去掉，那可能不是单独一个正则能做到的。现在切割器已经被我们搞得相当复杂了，维护性很差。在Mootools等引擎中，里面的正则表达式更加复杂，可能是用工具生成的。到了这个地步，我们就需要转换思路，将切割器改为一个函数处理。当然，它里面也少不了正则表达式。正则是处理字符串的利器 。

```
var reg_split =
/^[\w\u00a1-\uFFFF\-\*]+|[#.:][\w\u00a1-\uFFFF-]+(?:\([^\])*\))
?|\[[^\]]*\]|(?:\s*) [>+~,](?:\s*)|\s(?=[\w\u00a1-\uFFFF*#.[:])
|^\s+/;
var slim = /\s+|\s*[>+~,*]\s*$/
```

```
function spliter(expr) {
    var flag_break = false;
    var full = [];//这里放置切割单个选择器群组得到的词素，以","为界
    var parts = [];//这里放置切割单个选择器组得到的词素，以关系选择器为
界
    do {
        expr = expr.replace(reg_split, function(part) {
            if (part === ",") {//这个切割器只处理到第一个并联选择器
                flag_break = true;
            } else {
                if (part.match(slim)) {//对关系并联。通配符选择器两
边的空白进行处理
                    //对parts进行反转，因为div.aaa，反转后先处理.aaa
                    full = full.concat(parts.reverse(), part.re
place(/\s/g, ''));
                    parts = [];
                } else {
                    parts[parts.length] = part;
                }
            }
            return "";//去掉已经处理了的部分
        });
        if (flag_break)
            break;
    } while (expr)
    full = full.concat(parts.reverse());
    !full[0] && full.shift();//去掉开头第一个空白
    return full;
}
var expr = "  div  >  div#aaa,span"
console.log(spliter(expr));//["div",">","#aaa", "div"]
```

当然，这个相对于Sizzle1.8与Slick等引擎的切割器来说，不值一提。写好一个切割器，需要有非常深厚的正则表达式功力以及深层的知识包括自动机理论。

5.4.5 属性选择器对于空白字符的匹配策略

已经介绍过属性选择器的7种形态了，但属性选择器并没有这么简单。在W3C草案对属性选择器[att~=val]提到一个点，val不能为空白字符，否则比较值flag（flag为val与元素实际值的比较结果）总返回false。如果用querySelectorAll测试一下属性其他形态，我们会得到更多类似结果。

```
<!DOCTYPE html>
<html>
    <head>
        <title>属性选择器</title>
        <meta http-equiv="Content-Type" content="text/html; cha
rset=UTF-8">
        <script>
            window.onload =function(){
                console.log(document.querySelector("#test1[titl
e='']"));
                console.log(document.querySelector("#test1[titl
e~='']"));
                console.log(document.querySelector("#test1[titl
e|='']"));
                console.log(document.querySelector("#test1[titl
e^='']"));
                console.log(document.querySelector("#test1[titl
e$='']"));
                console.log(document.querySelector("#test1[titl
e*='']"));
                console.log("===================================
=========")
```

```
                console.log(document.querySelector("#test2[titl
e='']"));
                console.log(document.querySelector("#test2[titl
e~='']"));
                console.log(document.querySelector("#test2[titl
e|='']"));
                console.log(document.querySelector("#test2[titl
e^='']"));
                console.log(document.querySelector("#test2[titl
e$='']"));
                console.log(document.querySelector("#test2[titl
e*='']"));
            }
        </script>
    </head>
    <body>
        <div title="" id="test1"></div>
        <div title="aaa" id="test2"></div>
    </body>
</html>
```

运行结果如下：

```
日志: [object HTMLDivElement]
日志:null
日志: [object HTMLDivElement]
日志:null
日志:null
日志:null
日志: ========================================
日志:null
日志:null
日志:null
日志:null
日志:null
日志:null
```

换言之，只要val为空，除=或|=外，flag必为false。并且，对于非=，!=操作符，如果取得值为空白字符，flag也必为false。

```
//https://github.com/jquery/sizzle/blob/2.3.0/src/sizzle.js#L11
75-L1197
"ATTR": function( name, operator, check ) {
    return function( elem ) {
        var result = Sizzle.attr( elem, name );

        if ( result == null ) {
            return operator === "!=";
        }
        if ( !operator ) {
            return true;
        }

        result += "";

        return operator === "=" ? result === check :
            operator === "!=" ? result !== check :
            operator === "^=" ? check && result.indexOf( check
) === 0 :
            operator === "*=" ? check && result.indexOf( check
) > -1 :
            operator === "$=" ? check && result.slice( -check.l
ength ) === check :
            operator === "~=" ? ( " " + result.replace( rwhites
pace, " " ) + " " ).
            indexOf( check ) > -1 :
            operator === "|=" ? result === check || result.slic
e( 0, check.length + 1 ) ===
            check + "-" :
            false;
    };
},
```

5.4.6　子元素过滤伪类的分解与匹配

子元素过滤伪类是CSS3新增的一种选择器，比较复杂，这里单独说一下。首先，我们要将它从选择符中分离出来，这个一般由切割器搞定。然后我们用正则将伪类名与它小括号里的传参分解出来。下面介绍一下Icarus的做法。

```
var expr = ":nth-child(2n+1)",
    rsequence = /^([#\.:]|\[\s*]((?:[-\w]|[^\x00-\xa0]|\\.)+)/,
    rpseudo = /^\(\s*("([^"]*)"|'([^']*)'|[^\(\)]*(\([^\(\)]*\)
)?)\s*\)/, rBackslash = /\\/g,
    //这里把伪类从选择符里分解出来
    match = expr.match(rsequence); //[":nth-child",":", "nth-ch
ild"]
    expr = RegExp.rightContext;//用它左边的部分重写expr--> "(2n+1)
""
      key = (match[2] || "").replace(rBackslash, "");//去掉换行符
 key--> "nth-child"
switch (match[1]) {
    case "#":
        //ID选择器 略
        break;
    case ".":
        //类选择器 略
        break;
    case ":":
        //伪类 略
        tmp = Icarus.pseudoHooks[key];
        //Icarus.pseudoHooks里面放置我们所有能处理的伪类
        if (match = expr.match(rpseudo)) {
            expr = RegExp.rightContext;//继续取它左边的部分重写expr
            if ( !! ~key.indexOf("nth")) {//如果是子元素过滤伪类
                args = parseNth[match[1]] || parseNth(match[1])
;//分解小括号的传参
```

```
            } else {
                args = match[3] || match[2] || match[1]
            }
        }
        break
    default:
        //属性选择器 略
        break;
}
```

这里有个小技巧，我们需要不断把处理过的部分从选择符中去掉。一般的选择器引擎是使用`expr = expr.replace(reg, "")`进行处理。Icarus是巧妙地使用正则的RegExp.rightContext进行复写，将小括号里面的字符串取得后我们通过parseNTH进行加工，将数字1、4，单词even、odd、−n + 1等各种形态转换an+b的形态。

```
function parseNth(expr) {
    var orig = expr
    expr = expr.replace(/^\+|\s*/g, ''); //清除无用的空白
    var match = (expr === "even" && "2n" || expr === "odd" && "
2n+1" || !/\D/.test(expr) && "0n+" + expr || expr).match(/(-?)(
\d*)n([-+]?\d*)/);
    return parse_nth[orig] = {
        a: (match[1] + (match[2] || 1)) - 0,
        b: match[3] - 0
    };
}
```

parseNth是一个缓存函数，这样就能避免重复解析，提高引擎总体性能。

我们再来看一下an+b的匹配算法，a与b都是整数，可能是正数、负数、零。最简单的情形是a为零，b随意时。我们得到此元素的父节点下的所有子元素，即els = el.parentNode.children，那么匹配算法浓缩成`!!` 为`els[b-1]`。因为子元素过滤伪类中的传参是以1开始，因此需要减1。

如果a为1，b为0的情况，这也很简单，意味着所有孩子都匹配，直接返回true就行。如果b为0时，比如2n就是取孩子中的索引值为偶数的个体，假设索引值为index，那么(index + 1) % 2 == 0就能匹配出来，由此推断出公式为`(index + 1) % a == 0`。

再看b不为0的情况，比如:nth-child(3n+1)，一个拥有10个孩子的节点，它的索引值中第0、3、6、9的孩子被匹配，即(index + 1 − 1) % 3 == 0时被匹配，因此匹配公式为(index + 1 - b) % a == 0 。

不过当b > a时，又有新状况，比如:nth-child(3n+4)，还是那个拥有十个孩子的节点，这时只有索引值为3、6、9的孩子被匹配，第一个孩子(0 + 1 − 4) % 3也是等于零，这时我们需要判定商是否大于负数了。由于index总是要加1，那么我们一开始把起点弄成1，那么就省事多了。总结上面的分析，我们把索引值与b的差作为一个变量diff，那么匹配规则为diff % a == 0 && diff/ a >= 0 。

我们再来看nth-of-type，它其实就是对父元素的child进行分类，分类标签是元素的tagName。然

后只要目标元素的索引值（也是从1开始）匹配上面的公式就行了。

从性能上考量，并不是每种子元素过滤伪类都需要通过这种比较索引值的方式实现，我们可以变通一下。比如说，:first-child，我们看一下手头上的这个元素是否存在兄长，亦即**previousElementSibling**，存在就返回false过滤掉。:fast-child也是基于相同的思路进行过滤。:only-child，需要判定目标元素是否为其父亲的唯一一个子元素，需要同时判定是否存在**previousElementSibling**与**nextElementSibling**。当然，现实是残酷的，为了兼容旧版本IE，我们一般用previousSibling、nextSibling加nodeType是否为1进行判定。

但对于:nth-child, :nth-last-child, :nth-of-type, :nth-last-of-type就不行了，我们也不可能每匹配一

个元素就把它在兄弟中或tagName分组中的索引值计算一遍。因此以空间换时间这万能药再次被提出来了，我们将缓存仓库设置在父节点上，然后把它所有孩子的索引值或tagName分组一次性放在上面，就可以在一次查找过程中反应利用此缓存。至于如何实现可参看Icarus、Sizzle1.8+、Slice等引擎，代码比较长，在此不再贴出来。

5.5　Sizzle引擎

jQuery最大的特点是其选择器，jQuery1.3时开始列装其Sizzle引擎。Sizzle引擎与当时主流的引擎大不一样，人们说它是从右到左选择（虽然也不对，只能说大致方向如此），速度远胜当时的选择器。

Sizzle当时的几大特点如下。

（1）允许以关系选择器开头。

（2）允许取反选择器套取反选择器。

（3）大量的自定义伪类，比如位置伪类（:eq, :first, :even……）、内容伪类（:contains）、包含伪类（:has）、标签伪类（:radio, :input, :text, :file……）、可见性伪类（:hidden,:visible）。

（4）对结果进行去重，以元素在DOM树的位置进行排序，这样与未来出现的querySelector行为一致。

显然，一下子搞出这么多东西，不是一朝一夕的事，说明John Resig已经研发了很久。当时Sizzle的版本号为0.9.1，代码风格与jQuery库大不一样，非常整齐清晰。这个风格一直延续到jQuery1.7.2，Sizzle版本号也跟上为1.7.2。在jQuery1.8或者说Sizzle1.8中，它风格大变，首选里面的正则是通过编译得到的，以求更加准确，结构也异常复杂，开始走EXT那样的编译函数的路子，以求通过多种缓存手段提高查找速度和匹配速度。

由于Sizzle1.8加入编译机制后，代码变得非常复杂难懂，分析起来篇幅太长了，我们还是选用1.7.2来讲解Sizzle的主流程，如图5-3所示。

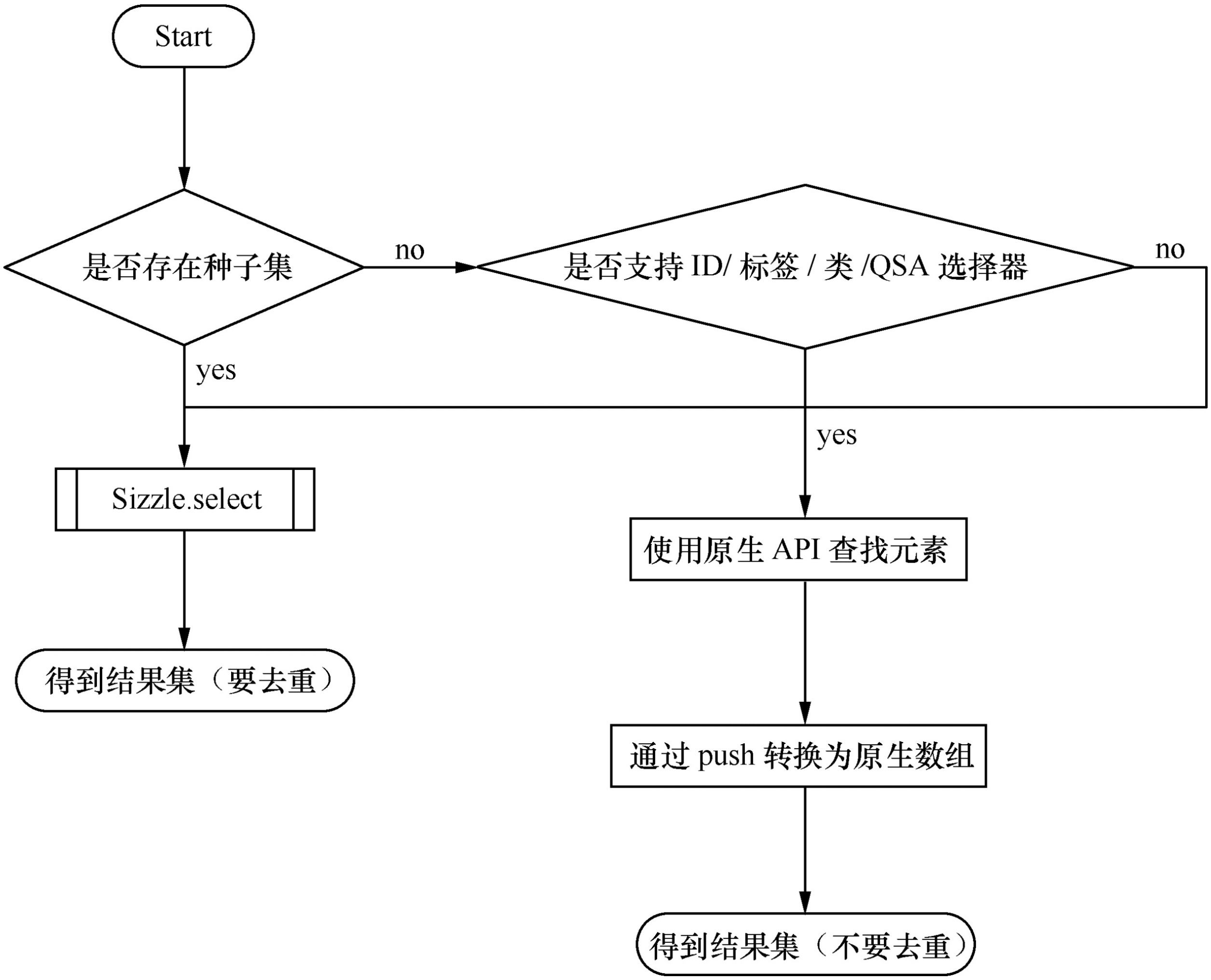


图5-3

Sizzle.select里的流程，如图5-4所示。

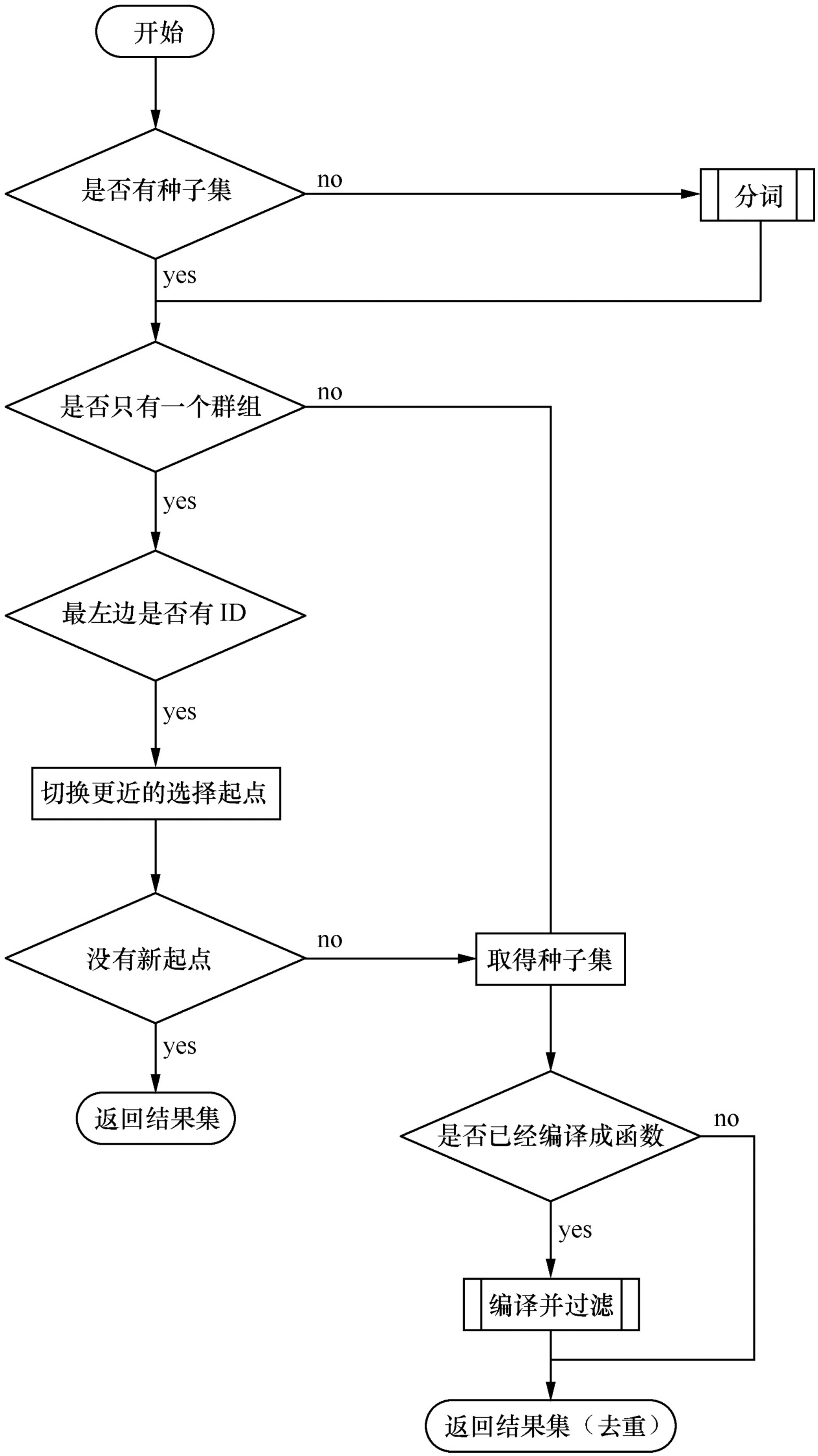

开始
是否有种子集
no
分词
yes
是否只有一个群组
no
yes
最左边是否有 ID
yes
切换更近的选择起点
没有新起点
no
取得种子集
yes
返回结果集
是否已经编译成函数
no
yes
编译并过滤
返回结果集（去重）

图5-4

Sizzle的整体结构如下。

（1）Sizzle主函数，里面包含选择符的切割，内部循环调用主查找函数，主过滤函数，最后是去重过滤。

（2）其他辅助函数，如uniqueSort、matches、matchesSelector。

（3）Sizzle.find主查找函数。

（4）Sizzle.filter主过滤函数。

（5）Sizzle.selectors包含各种匹配用的正则、过滤用的正则、分解用过的正则、预处理函数、过滤函数。

（6）根据浏览器的特征设计makeArray、

sortOrder、contains等方法。

（7）根据浏览器的特征重写Sizzle.selectors中的部分查找函数、过滤函数、查找次序。

（8）若浏览器支持querySelectorAll，那么用它重写Sizzle，将原来的Sizzle作为后备方案包裹在新Sizzle里面。

（9）其他辅助函数，如isXML、posProcess。

在jQuery 1.8后，加入tokenize（就是第一步提到选择符切割），然后加入编译机制，编译机制实际上就是根据我们切割好的字符的类型得到一大堆匹配函数，然后将它们转换为一个柯里化函数**superMatcher**，然后再将这些字符作为传参来求值。上面的匹配函数就包括了jQuery1.7的主查找函数与主过滤函数。

```
<a>https://github.com/jquery/sizzle/blob/1.7.2/sizzle.js</a>
```

```
var Sizzle = function(selector, context, results, seed) {
//通过短路运算符，设置一些默认值
    results = results || [];
context = context || document;
//备份，因为context会被改写，如果出现并联选择器，就无法区别当前节点是对应哪一个content

    var origContext = context;
//上下文对象必须是元素节点或文档对象
    if (context.nodeType !== 1 && context.nodeType !== 9) {
        return [];
    }
//选择符必须是字符串，且不能为空
    if (!selector || typeof selector !== "string") {
        return results;
    }

    var m, set, checkSet, extra, ret, cur, pop, i,
            prune = true,
            contextXML = Sizzle.isXML(context),
            parts = [],
            soFar = selector;
//下面是切割器的实现，每次只处理到并联选择器，extra留给下次递归自身时作传参
//不过与其他引擎的实现不同的是，它没有一下子切成选择器，而是切成选择器组与关系选择器的集合
//比如body div > div:not(.aaa),title
//将会得到parts数组：["body","div",">", "div:not(.aaa)"]
//后代选择器虽然被忽略了，但在循环这个数组时，它默认每两个选择器组中一定夹着关系选择器
//不存在就放在后代选择器到那个位置上
    do {
        chunker.exec(""); //这一步主要是将chunker的lastIndex重置，当然直接设置chunker.
                            //lastIndex效果也一样
        m = chunker.exec(soFar);
        if (m) {
            soFar = m[3];
            parts.push(m[1]);
            if (m[2]) { //如果存在并联选择器，就中断
                extra = m[3];
                break;
            }
        }
    } while (m);
// ……略
```

```
}
```

接下来有许多分支，分别是对ID与位置伪类进行优化的。我们暂时跳过它们，看另外几个重要概念：查找函数、种子集、映射集。虽然有的之前介绍过，但是这里是结合Sizzle源码讲解的。

查找函数就是Sizzle.selecters.find下的几种函数，常规情况下有ID、TAG、NAME三个，如果浏览器支持getElementsByClassName，还会有Class函数。正如笔者前面所介绍的那样，getElementById、getElementsByName、getElementsByTagName和getElementsByClassName不能完全信任它们，即便是标准浏览器都会有bug，因此四大查找函数都做了一层薄薄的封装，不支持则返回undefined，其他则返回数组或NodeList。

种子集，就是通过最右边的选择器组得到的元素集合，比如说div.aaa span.bbb，最右边的选择器组就是"span.bbb"。这时引擎会根据浏览器的支持情况选择getElementsByTagName或getElementsClassName得到一组元素，然后再通过className或tagName进行过滤，这时得到的集合就是种子集。Sizzle的变量名seed就体现了这一点。

映射集，Sizzle源码的变量名为checkSet。当我们取得种子集后，不动种子集，而是将种子集复制一份出来，这就是映射集。种子集是由一个选择器组选出来的，这时选择符不为空，必然往左就是关系选择器。关系选择器会让引擎去选取其兄长或父亲（具体操作见Sizzle.selectors.relative下的四大函数），把这些元素置换到候选集对等的位置上。然后到下一个选择器组时，就是纯过滤操作。主过滤函数Sizzle.filter会调用Sizzle.seletors下N个过滤函数对这些元素进行检测，将不符合的元素替换为

false。因此到最后要去重排序时，映射集是一个包含布尔值与元素节点的数组（true值也在上面步骤中产生）。

让我们继续看源码。种子集是分两步筛选出来的。首先，通过Sizzle.find得到一个大体的结果。然后通过Sizzle.filter，传入最右的那个选择器组剩余的部分作参数，缩少范围。

```
//这是针对最左边的选择器组存在ID做出的优化
var ret = Sizzle.find(parts.shift(), context, contextXML);
context = ret.expr ? Sizzle.filter(ret.expr, ret.set)[0] : ret.
set[0];

ret = seed ? {
    expr: parts.pop(),
    set: makeArray(seed)
    //这里会对~,+进行优化,直接取它的上一级做上下文
    //处理一个上下文对象胜过对付N个上下文
} : Sizzle.find(parts.pop(),parts.length === 1 &&
     (parts[0] === "~" || parts[0] === "+") && context.parentNo
de ?
    context.parentNode : context, contextXML);

set = ret.expr ? Sizzle.filter(ret.expr, ret.set) : ret.set;
```

我们是先取span还是取.aaa呢？这里有个准

则，即确保我们后面的映射集最小化。直白地说，映射集里面的元素越少，那么调用过滤函数的次数就越少，调用函数的次数越少，说明进入另一个函数作用域所造成的能耗就越少，从而整体提高引擎的选择速度。为了达到此目的，这里做了个优化，原生选择器的调用顺序被放到一个叫Sizzle.slectors.order的数组中。对于陈旧的浏览器，其顺序为ID、NAME、TAG；对于支持getElementsByClassName的浏览器，其顺序为ID、CLASS、NAME、TAG。因为ID至多返回一个元素节点，className与样式息息相关，不是每个元素都有这个类名。name属性带来的限制可能比className更大，但用到的机率比较少，而tagName可排除的元素则更少了。那么Sizzle.find就会根据上面的数组，取得它的名字依次调用Sizzle.leftMatch下对应的正则，从最右的选择器组中切下需要的部分，将换行符处理掉，通过四大查找函数得到一个粗糙的节点集合。如果运气太差，

碰到如[href=aaa]:visible这样的选择符，那么只有把文档中的所有节点作为结果返回。

```
Sizzle.find = function(expr, context, isXML) {
    var set, i, len, match, type, left;

    if (!expr) {
        return [];
    }

    for (i = 0, len = Expr.order.length; i < len; i++) {
        type = Expr.order[i];
        //取得正则,匹想出需要的ID、CLASS、NAME、TAG
        if ((match = Expr.leftMatch[type].exec(expr))) {
           left = match[1];
           match.splice(1, 1);
           //处理换行符
           if (left.substr(left.length - 1) !== "\\") {
                match[1] = (match[1] || "").replace(rBackslash,
 "");
                set = Expr.find[type](match, context, isXML);
                //如果不为undefined,那么去掉选择器组中用过的部分
                if (set != null) {
                    expr = expr.replace(Expr.match[type], "");
                    break;
                }
           }
        }
    }

    if (!set) { //没有,寻找该上下文对象的所有子孙
        set = typeof context.getElementsByTagName !== "undefine
d" ?
        context.getElementsByTagName("*") : [];
    }

    return {
        set: set,
        expr: expr
    };
};
```

经过主查找函数处理后，我们得到一个初步的结果，这时最右边的选择器组可能还有残余。比如`div span.aaa`可能余下`div span`，`div .aaa.bbb`或者可能余下`div .bbb`，这个转交主过滤函数Sizzle.filter函数处理。它有两种不同的功能，一是不断缩小集合的个数，构成种子集返回；另一种是将原集合中不匹配的元素置换为false，这个根据它的第三个传参inplace而定。

```
Sizzle.filter = function(expr, set, inplace, not) {
    //用于生成种子集或映射集,这视第三个参数而定
    //expr: 选择符
    //set:  元素数组
    //inplace:undefined, null时进入生成种子集模式,true时进入映射集模
式
    //not:  一个布尔值,来源自取反选择器
    var match, anyFound,
    type, found, item, filter, left,
    i, pass,
    old = expr,
        result = [],
        curLoop = set,
        isXMLFilter = set && set[0] && Sizzle.isXML(set[0]);

    while (expr && set.length) {
        for (type in Expr.filter) { //ID,TAG,CLASS,TAG,CHILD,PO
S,PSEUDO
            if ((match = Expr.leftMatch[type].exec(expr)) != nu
```

```
ll && match[2]) {
                //切割出相应的字符串,作为传参放进filter里面
                filter = Expr.filter[type];
                left = match[1];
                //ID -->    ["#aaa","","aaa"]
                //CLASS --> [".aaa","","aaa"]
                //TAG-->    ["div","","div"]
                //ATTR-->   ["[aaa=ggg]", "", "aaa", "^=", unde
fined, undefined, "ggg"]
                //CHILD--> [":nth-child(even)", "", "nth-child"
, "", "even"]
                //POS-->    [":eq(2)", "", "eq", "2"]
                //PSEUDO-->[":not(.aaa)", "", "not", "", ".aaa"
]
                anyFound = false;
                match.splice(1, 1);
                if (left.substr(left.length - 1) === "\\") {
                    continue;
                }
                if (curLoop === result) {
                    result = [];
                }
                if (Expr.preFilter[type]) {
                    match = Expr.preFilter[type](match, curLoop
, inplace, result, not,
                    isXMLFilter);
                    //这里会对传参进行加工,比如#aaa得到aaa, .bbb得到b
bb
                    //出于优化需要,它会在Expr.preFilter.CLASS进行过
滤
                    //而不用等于Expr.filter.ClASS
                    //另外,针对CHILD, POS,它会两入进入这个循环,因为它
们同时也能被
                    //Expr.leftMatch.PSEUDO匹配,但它不想被Expr.fil
ter.PSEUDO处理
                    //于是直接continue
                    if (!match) { //CLASS
                        anyFound = found = true;
                    } else if (match === true) { //CHILD, POS
                        continue;
                    }
                }
                if (match) {
                    //curLoop为一个映射集,里面包含false, true
                    for (i = 0;
```

```
                    (item = curLoop[i]) != null; i++) {
                        if (item) {
                            found = filter(item, match, i, curLo
op);
                            pass = not ^ found;
                            //在映射集模式下,将不匹配的元素置换为false
                            if (inplace && found != null) {
                                if (pass) {
                                    anyFound = true;
                                } else {
                                    curLoop[i] = false;
                                }
                                //否则result为我们的种子集,把匹配者放
进去

                            } else if (pass) {
                                result.push(item);
                                anyFound = true;
                            }
                        }
                    }
                }

                if (found !== undefined) {
                    if (!inplace) {
                        curLoop = result; //重写种子集为curLoop
                    }
                    //削减选择符直到变为空字符串
                    expr = expr.replace(Expr.match[type], "");
                    if (!anyFound) {
                        return [];
                    }

                    break;
                 }
            }
        }

        //如果到最后正则表达式也不能改动选择符，说明它有问题
        if (expr === old) {
            if (anyFound == null) {
                Sizzle.error(expr);
            } else {
                break;
            }
        }
```

```
        old = expr;
    }
    return curLoop;
};
```

待到我们把最右边的选择器组的最一个字符都去掉后，种子集宣告完成，然后处理下一个选择器组，并将种子集复制一下，生成映射集。在关系选择器4个对应函数，它们位于Sizzle. selectors. relative 命名空间下，只是将映射集里面的元素置换为它们的兄长父亲，个数是不变的。因此映射集与种子集的数量总是相当。另外，这4个函数内部也在调用Sizzle.filter函数，它的inplace参数为true，走映射集的逻辑。

```
while (parts.length) {
    cur = parts.pop(); //取得关系选择器
    pop = cur;
    if (!Expr.relative[cur]) {
        cur = ""; //如果不是则默认为后代选择器
    } else {
        pop = parts.pop(); //取得后代选择器前面的子选择器群集
    }
    if (pop == null) {
        pop = context;
    }
    Expr.relative[cur](checkSet, pop, contextXML); //根据其他4种
迭代器改变映射集里面的元素
```

```
    //得到诸如[ [object HTMLDivElement] ,false, false, [object HTMLSpanElement] ]的集合
}
```

最后一步就是根据映射集甄选候选集。

```
for (i = 0; checkSet[i] != null; i++) {
    if (checkSet[i] && checkSet[i].nodeType === 1) {
        results.push(set[i]);
    }
}
```

如果存在并联选择器，那就再调用Sizzle主函数，把得到的两个结果合并去重。

```
if (extra) {
    Sizzle(extra, origContext, results, seed);
    Sizzle.uniqueSort(results);
}
```

这就是主流程。下面将是根据浏览器的特性进行优化或调整的部分。比如IE6、IE7下的getElementById有bug，需要重写Expr.find.ID与Expr.filterID。IE6～IE8下，Array.prototype.slice.call

无法切割NodeList，需要重写makeArray。IE6～IE8，getElementsByTagName(“*”)会混杂注释节点，需要重写Expr.find.TAG。如果浏览器支持querySelectorAll，那么需要重写个Sizzle。

```
if (document.querySelectorAll) {
    (function() {
        var oldSizzle = Sizzle,
            div = document.createElement("div"),
            id = "__sizzle__";

        div.innerHTML = "<p class='TEST'></p>";
        //Safari在怪异模式下querySelectorAll不能工作,中止重写
        if (div.querySelectorAll && div.querySelectorAll(".TEST
").length === 0) {
            return;
        }

        Sizzle = function(query, context, extra, seed) {
            //这里省略N行
            //更好的利用querySelector实现的选择器引擎
            return oldSizzle(query, context, extra, seed);
        };
        //将原来的方法重新绑定到新Sizzle函数上
        for (var prop in oldSizzle) {
            Sizzle[prop] = oldSizzle[prop];
        }

        // release memory in IE
        div = null;
    })();
}
```

一旦用上querySelectorAll, Sizzle的性能就飕飕

上去了！

5.6 总结

要实现兼容IE6的选择器引擎是非常难的，特别是要兼顾性能及支持CSS3伪类，就是难上加难。在querySelectorAll被普遍支持的今日，特别是移动端项目，根本没有必要用Sizzle这样专业的引擎。zepto.js就很取巧地直接用querySelectorAll，只有几行代码，但Sizzle则是1900行之巨。如果一定要说学习写一个选择器有什么意义，那就是各种设计模式的实验场了！Sizzle源码可谓是鬼斧神工，完全读通的话，你的JavaScript功力会提升几个档次。

如果一定要兼容IE6，但你又用不上什么复杂的选择器，那么就试试下面这个迷你选择器。最后作为“饭后甜点”奉给大家。

```
function $(query) {
    var res = []
    if (document.querySelectorAll) {
        res = document.querySelectorAll(query)
    } else {
```

```
        var firstStyleSheet = document.styleSheets[0] || docume
nt.createStyleSheet()
        query = query.split(',')
        for(var i = 0, len = query.length; i < len; i++) {
          firstStyleSheet.addRule(query[i], 'Hack:ie')
        }
        for (var i = 0, len = document.all.length; i < len; i++
) {
          var item = document.all[i]
          item.currentStyle.Hack && res.push(item)
        }
        firstStyleSheet.removeRule(0)
    }
    var ret = []
    for(var i = 0, len = res.length; i < len; i++){
        ret.push(res[i])s
    }
    return ret
}
```

---

[1]　现在所有新的浏览器都支持这个特性。

[2]　IE6-8支持使用<comment>我是注释</comment>、<%我是注释%>、<！我是注释>（这个所有浏览器都支持，但我们通常是被教导使用<！——前后都有两个横杆——>）这3种方式定义注释节点。因此，我们不觉意间就创建了许多注释节点。

[3]　这个选择器及其不稳定，已经换了3种语法了。

[4]　sort各个浏览器是如何实现的，ecma并没有具体规定，对稳定性也没有要求：15.4.4.44 Array.prototype.sort(comparefn)

The elements of this array are aorted.The sort is not necessarily stable (that is,elements that compare equal do not necessarily remain in their original order).

由此，浏览器厂商便各显神通，采用不同的排序算法实现sort方法。各个算法的稳定性不同而导致了结果的差异。Firefox2采取了不稳定的堆排序，firefox3采用了稳定的归并排序，ie速度较慢，具体算法不明，可能为冒泡或者插入排序，而chrome则为了最大效率，采用了两种算法：

Chrome use quick sort for a large dataset(> 22),and for a smaller dataset (< =22) chrome use insert sort but modified to use binary search to find insert point,so

traditionally insert sort is stable,but chrome make it faster and unstable,in conclusion chrome’ s sort is quick and unstable

第6章　节点模块



DOM操作占我们前端工作的很大一部分，其中元素节点的操作又占其50%以上。由于MVVM的流行，让用户直接操作DOM的机会越来越少，因此之前jQuery的链式操作我将全部删掉，新版本将集中介绍如何实现CRUD。没错，CRUD就是数据

库经常所说的那个CRUD，我们可以将整个DOM树作为一个大的数据表，然后对它们进行增删改查。由于查找元素已经在第5章介绍了，本章实际只有创建、更新与删除，及其他一些常用功能讲解。

6.1 节点的创建

浏览器提供了多种手段创建API，从流行度来看，依次是document.createElement、innerHTML、insertAdjacentHTML、createContextualFragment。

document.createElement基本不用说什么，它传入一个标签名，然后返回此类型的元素节点，并且对于浏览器还不支持的标签类型，它也能成功返回，这成了后来IE6～IE8支持HTML5新标签的救命稻草。在IE6～IE8中，它还有一种用法，能允许用户连同属性一起生成，比如document.createElement("<div id=aaa></div>")。此法常见于生成带name属性的input与iframe元素，因为**IE6～IE7**下这两种元素的**name**属性是只读的，不能修改。

```
//http://thunderguy.com/semicolon/2005/05/23/setting-the-name-a
ttribute-in-internet-explorer/
```

```
function createNamedElement(type, name) {
    var element = null;
    // Try the IE way; this fails on standards-compliant browse
rs
    try {
        element = document.createElement('<' + type + ' name="'
 + name + '">');
    } catch (e) {
    }
    if (!element || element.nodeName != type.toUpperCase()) {
        // Non-IE browser; use canonical method to create named
 element
        element = document.createElement(type);
        element.name = name;
    }
    return element;
}
```

innerHTML本来是IE的私有实现，现在遍地开花了。JQuery 1.0就开始发掘innerHTML的潜能，这不但是因为innerHTML的创建效率至少比createElement高2～10倍不等，还因为innerHTML能一下子生成一大堆节点。这与jQuery推崇批量操作的宗旨不谋而合。但innerHTML存在兼容性问题，比如**IE** 会对用户字符串进行 **trimLeft** 操作，本意是智能去掉无用的空白，但Firefox等则认为要忠于用户输入，对应位置要生成文本节点。

```
window.onload = function() {
    var div = document.createElement("div");
    div.innerHTML = " <b>1</b><b>2</b> "
    alert(div.childNodes.length); //IE6～IE8弹出3，其他4
    alert(div.firstChild.nodeType) //IE6弹出1，其他3
}
```

IE 下有些元素节点的 **innerHTML** 是只读的，重写innerHTML会报错，这就导致我们在动态插入节点时不能不转求appendChild、insertBefore来处理。下面出自MSDN：

http://msdn.microsoft.com/en-us/library/ms533897(VS.85).aspx The property is read/write for all objects except the following, for which it is read-only: COL, COLGROUP, FRAMESET, HEAD, HTML, STYLE, TABLE, TBODY, TFOOT, THEAD, TITLE, TR

IE 的 **innerHTML** 会忽略掉 **no-scope element**。no-scope element是IE的内部概念，隐藏得很深，仅

在MSDN中说明注释节点是no-scope element，或在social.msdn.microsoft.com官方论坛中透露一点内容——script与style也是no-scope elements。经过社区这么多年的发掘，大致确认注释、style、script、link、meta、noscript等表示功能性的标签为no-scope element。想要用innerHTML生成它们，必须在它们之前加上一些东西，比如文字或其他标签。

```
window.onload = function() { //请在IE6～IE8下测试
    var div = document.createElement("div");
    div.innerHTML = '<meta http-equiv="X-UA-Compatible" content
="IE=9"/>';
    alert(div.childNodes.length);
    div.innerHTML = 'X<meta http-equiv="X-UA-Compatible" conten
t="IE=9"/>';
    alert(div.childNodes.length);
};
```

另一个众所周知的问题是**innerHTML** 不会执行 **script** 标签里面的脚本 。其实也不尽然，如果浏览器支持script标签的defer属性，它就能执行脚本。这个特性检测比较难做，因此像jQuery直接用正则把它里面的内容抽取出来，然后全局eval了。

avalon则采取另一种策略，反正innerHTML赋值后已经将它们转换成节点，那么再将它们取出来用`document.createElement("script")`生成的节点代替就行了。

最后一个问题就是有的标签不能单独作为div的子元素，比如td、th元素，需要最外面包几层，才能放到innerHTML中解释，否则浏览器会当作普通的文本节点生成。这个是jQuery团队发现的，现在所有框架都使用此技术生成节点。如果把这些特殊的标签比作是“胚胎”，那么孵化它们出来的那些父元素就是“胎盘”。在W3C规范中，它们都是这样一组组地分成不同的模块，如表6-1所示。

表**6-1**

胚胎	胎盘
area	map
param	object

col	colgroup
legend	fieldset
option,optgroup	select
thead,tfoot,tbody,colgroup,caption	table
tr,	table
td, th	tr

一直以来，人们都是使用完整的闭合标签来包裹这些特殊标签，直到人们发现浏览器会自动补全闭合标签。

```
window.onload = function() {
    var div = document.createElement("div");
    div.innerHTML = '<table><tbody><tr></tr></tbody></table>';//手动闭合标签
    alert(div.getElementsByTagName("tr").length);//1
    div.innerHTML = '<table><tbody><tr></tr>';//让浏览器自动处理
    alert(div.getElementsByTagName("tr").length);//1
}
```

能自动补完结束标签的元素有body、

colgroup、dd、dt、head、html、li、optgroup、option、p、tbody、td、tfoot、th、thead、tr。浏览器这种机制显然是为了竞争的需要，从而吸引开发者转向自己的阵营。对于浏览器而言，根据上下文补完标签不是什么难事，加之，这能有效减少页面的大小，在网速奇慢的年代是一个优化。同时不写结束标签能避免文本节点出现在元素前后，因此也能减少页面上节点的总体数量。

但现在已经不推荐这样做，浏览器只会固守规则，少写结束标签，很容易引起错误镶嵌。xhtml布道者就是抓住这一点死命抨击HTML4。但在JavaScript框架内部，由于是把标签限制在一个div内，由经验丰富的JSer来处理，因此还是可以利用的。

InsertAdjacentHTML 是dhtml的产物，这也是IE的私有实现。比起其他API，它具有灵活的插入方

式。你可以插入到一个元素内部的最前面（afterBegin）、内部的最后面（beforeEnd）、这个元素的前面（beforeBegin）、后面（afterEnd）。它们一一对应jQuery的prepend、append、before、after。因此用它来构造这几个方法，代码量会大大减少。但不巧的是，insertAdjacentHTML要我们的字符串同样遵守HTML的套嵌规则。在IE下，它在td、th等元素内部插入新节点还是报错，理由同innerHTML。不过，若我们能提早判定用户字符串没有需要套嵌的元素、没有no-scope元素的话，那么在插入操作中它还是很有用的。

jQuery相关的操作是先经由append方法进入domManip方法，再到buildFragment方法，再到clean方法，这么复杂才完成。其间有字符串再加工、script内容抽取、innerHTML序列化、文档碎片对象生成、插入DOM、全局eval这么多步骤。在最

理想的情况，我们可以用一个insertAdjacentHTML搞定。insertAdjacentHTML的兼容性如表6-2所示。

表**6-2**

浏览器	**Chrome**	**Firefox**	**IE**	**Opera**	**Safari(webkit)**
版本	1	8	4	7	4(527)

如果浏览器不支持 insertAdjacentHTML，那么我们可以用下面介绍到的 createContextual Fragment来模拟。

```
if (typeof HTMLElement !== "undefined" &&
 !HTMLElement.prototype.insertAdjacentElement) {
    HTMLElement.prototype.insertAdjacentElement = function(wher
e, parsedNode) {
            switch (where.toLowerCase()) {
            case 'beforebegin':
                this.parentNode.insertBefore(parsedNode, this)
                break;
            case 'afterbegin':
                this.insertBefore(parsedNode, this.firstChild);
                break;
            case 'beforeend':
                this.appendChild(parsedNode);
                break;
            case 'afterend':
                if (this.nextSibling)
                this.parentNode.insertBefore(parsedNode, this.n
extSibling);
```

```
                else this.parentNode.appendChild(parsedNode);
                break;
        }
    }
    HTMLElement.prototype.insertAdjacentHTML = function(where, 
htmlStr) {
        var r = this.ownerDocument.createRange();
        r.setStartBefore(this);
        var parsedHTML = r.createContextualFragment(htmlStr);
        this.insertAdjacentElement(where, parsedHTML)
    }
    HTMLElement.prototype.insertAdjacentText = function(where, 
txtStr) {
        var parsedText = document.createTextNode(txtStr)
        this.insertAdjacentElement(where, parsedText)
    }
}
```

createContextualFragment是Firefox推出的私有实现，它是Range对象的一个实例方法，相对于insertAdjacentHTML直接将内容插入到DOM树。createContextualFragment则是允许我们将字符串转换为文档碎片，然后再由你决定插入到哪里。在著名的emberjs中，如果支持Range，那么它的html、append、prepend、after等方法都用createContextualFragment与deleteContents实现。createContextualFragment与insertAdjacentHTML一

样，要字符串遵守HTML的套嵌规则。

此外，我们还可以用document.write来创建内容，但我们动态添加节点时多发生在DOM树建完之后，因此不太合适，这里就不展开了。

最后要隆重介绍的template标签，它是一个天然的html parser，能将我们赋予它的字符串直接 转换为文档碎片！

```
var a = document.createElement('template')
a.innerHTML = '<div></div><div></div>'
console.log(a.content)

var a = document.createElement('template')
a.innerHTML = '<div></div><div></div>'
Console.log(a.content)
▼ #document-fragment
    <div></div>
    <div></div>
```

最后封上avalon的parseHTML方法，也是所有主流库都拥有的一个方法，将一段HTML转换为一个文档碎片。

```
https://github.com/RubyLouvre/avalon/blob/2.0.2/src/dom/html/mo
dern.js
var Cache = require('../../seed/cache')
var fixScript = require('./fixScript')
var tagHooks = new function () {
//这里按照上面提到的胚胎与胎盘的关系，将需要套嵌使用的母元素预先准备好
    avalon.shadowCopy(this, {
        option: document.createElement('select'),
        thead: document.createElement('table'),
        td: document.createElement('tr'),
        area: document.createElement('map'),
        tr: document.createElement('tbody'),
        col: document.createElement('colgroup'),
        legend: document.createElement('fieldset'),
        _default: document.createElement('div'),
        'g': document.createElementNS('http://www.w3.org/2000/s
vg', 'svg')
    })
    this.optgroup = this.option
    this.tbody = this.tfoot = this.colgroup = this.caption = th
is.thead
    this.th = this.td
}
//生成SVG套嵌使用的母元素
var svgHooks = {
    g: tagHooks.g
}
String('circle,defs,ellipse,image,line,path,polygon,polyline,re
ct,symbol,text,use').
replace(avalon.rword, function (tag) {
    svgHooks[tag] = tagHooks.g //处理SVG
})

var rtagName = /<([\w:]+)/
var rxhtml = /<(?!area|br|col|embed|hr|img|input|link|meta|para
m)(([\w:]+)[^>]*)\/>/ig

var rhtml = /<|&#?\w+;/
var htmlCache = new Cache(128)
var templateHook = document.createElement('template')
//如果浏览器不支持HTML5 template元素
if (!/HTMLTemplateElement/.test(tempateTag)) {
    templateHook = null
    avalon.shadowCopy(tagHooks, svgHooks)
}
```

```
avalon.parseHTML = function (html) {
    var fragment = document.createDocumentFragment(), firstChil
d
    if (typeof html !== 'string') {
        return fragment
    }
    if (!rhtml.test(html)) {
        fragment.appendChild(document.createTextNode(html))
        return fragment
    }
    html = html.replace(rxhtml, '<$1></$2>').trim()
    var hasCache = htmlCache.get(html)
    if (hasCache) {
        return hasCache.cloneNode(true)
    }
    var tag = (rtagName.exec(html) || ['', ''])[1].toLowerCase(
)
    var wrapper = svgHooks[tag], firstChild
    if (wrapper) {//svgHooks
        wrapper.innerHTML = html
    } else if (templateHook) {//templateHook
        templateHook.innerHTML = html
        wrapper = templateHook.content
    } else {//tagHooks
        wrapper = tagHooks[tag] || tagHooks._default
        wrapper.innerHTML = html
    }
    fixScript(wrapper)
    if (templateHook) {
        fragment = wrapper
    } else {// 将wrapper上的节点转移到文档碎片上!
        while (firstChild = wrapper.firstChild) {
            fragment.appendChild(firstChild)
        }
    }
    if (html.length < 1024) {
        htmlCache.put(html, fragment.cloneNode(true))
    }
    return fragment
}
```

Cache是一个带长度限制的缓存体，大致如下。

```
function Cache(size) {
    var keys = []
    var cache = {}
    this.get = function (key) {
        return cache[key+' ']
    }
    this.put = function (key, value) {
        //加上' '是用于处理IE6-8的toString, valueOf方法
        if (keys.push(key + ' ') > size) {
            delete cache[ keys.shift() ]
        }
        return (cache[ key + ' ' ] = value)
    }
    return this
}
```

fixScript顾名思义，用于修复innerHTML生成的script节点，不会发出请求与执行text属性的行为。

```
var scriptNode = document.createElement('script')
var scriptTypes = avalon.oneObject(['', 'text/javascript', 'tex
t/ecmascript',
    'application/ecmascript', 'application/javascript'])

function fixScript(wrapper) {
    var els = wrapper.getElementsByTagName('script')
    if (els.length) {
        for (var i = 0, el; el = els[i++]; ) {
            if (scriptTypes[el.type]) {
```

```
                //以偷龙转凤方式恢复执行脚本功能
                var neo = scriptNode.cloneNode(false) //FF不能省
略参数
                Array.prototype.forEach.call(el.attributes, fun
c tion (attr) {
                    if (attr && attr.specified) {
                        neo[attr.name] = attr.value //复制其属性
                        neo.setAttribute(attr.name, attr.value)
                    }
                }) // jshint ignore:line
                neo.text = el.text
                el.parentNode.replaceChild(neo, el) //替换节点
            }
        }
    }
}

module.exports = fixScript
```

6.2　节点的插入

从原生API单纯的字面来看，浏览器最初提供了insertBefore与appendChild这两个方法。还有一个是通过替换已有节点的“插入方式”，replaceChild。然后人们觉得不够用，于是一般的工具库还创建了这么一个方法。

```
function insertAfter(newElement, targetElement) {
     var parent = targetElement.parentNode;
     if (parent.lastChild == targetElement) {
         parent.appendChild(newElement);
     } else {
         parent.insertBefore(newElement, targetElement.nextSibl
ing);
     }
}
```

在微软的DHTML推销过程中，又造出了insertAdjacentXXX三组方法。

- insertAdjacentHTML。
- insertAdjacentText。

- insertAdjacentElement。

参数都相同，第1个是插入的位置，第2个是插入的内容（insertAdjacentHTML与insertAdjacentText要求传入字符串，insertAdjacentElement要求传入节点）。重点是第1个插入的位置，提供了4种方式（见图6-1）。

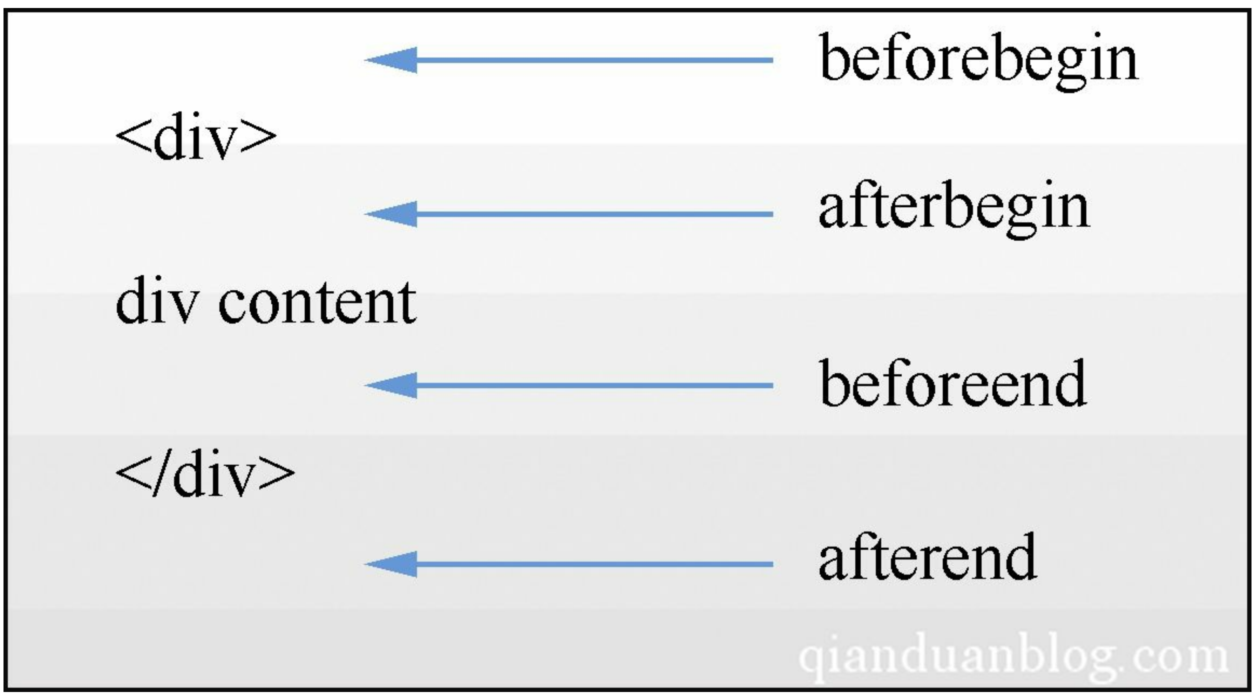


图6-1

（1）beforebegin：插入到标签开始前。

（2）afterbegin：插入到标签开始标记之后。

（3）beforeend：插入到标签结束标记前。

（4）afterend：插入到标签结束标记后。

这为以后jQuery的before、prepend、append、after方法的设计提供了一个样板。到后来，由于这几个方法太受欢迎，W3C在DOM4中决定原生支持它们[1]，参数可以是字符串与DOM节点，如图6-2所示。

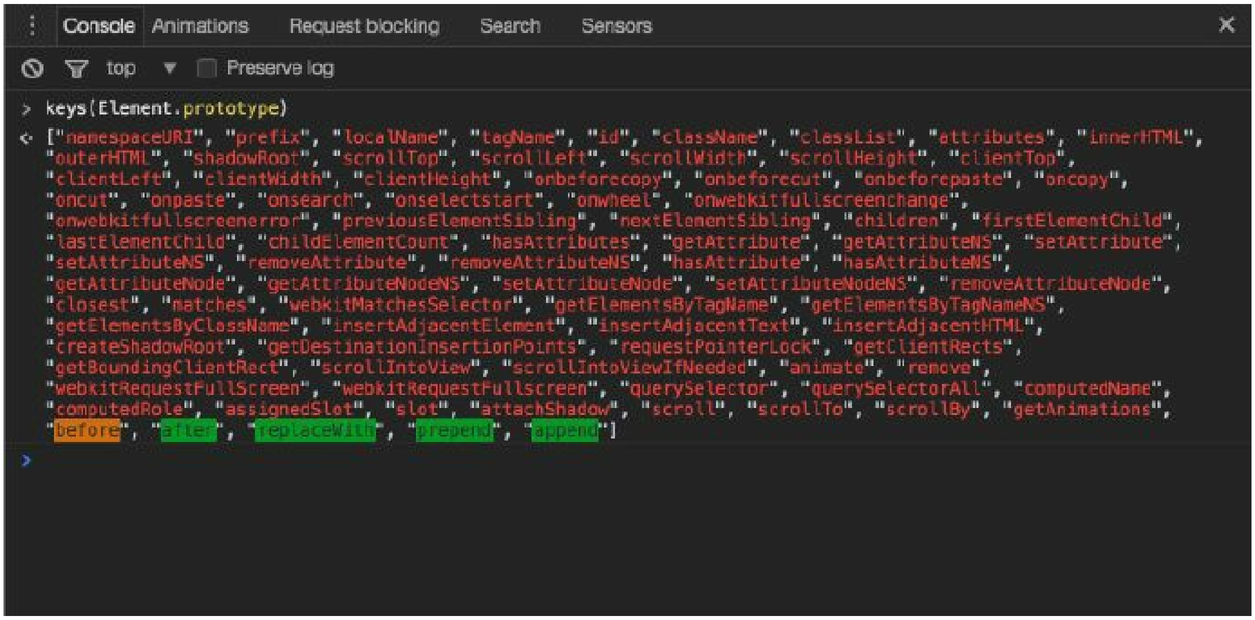


图6-2

除此之外，jQuery还提供了wrap、wrapAll、wrappInner这3种特殊的插入操作。

wrap为当前元素提供了一个父节点，此父节点将动态插入原节点的父亲底下。这个我们可以轻松在IE下用neo.applyElement (old, "outside")实现。

wrapAll则是为一堆元素提供一个共同的父节点，插入到第一个元素的父亲底下，其他元素再统统挪到新节点底下。

wrappInner是为当前元素插入一个新节点，然后将它之前的孩子挪到新节点底下。这个我们可以轻松在IE下用neo.applyElement (old, "inside")实现。从上面的描述来看，applyElement真是很强大的，可以在标准浏览器扩展一下，让它应用更广！

```
if (!document.documentElement.applyElement && typeof HTMLElement !== "undefined") {
```

```
   //实现IE only的removeNode
    HTMLElement.prototype.removeNode = function(deep) {
        //deep参数决定是否只剩除此节点，还是将其下级的所有子孙都一起删掉
        //如果只删目标节点（deep=false或不传参）
        //那么将其子孙全部上挪到目标节点的位置上
        var parent = this.parentNode
        var childNodes = this.childNodes
        var fragment = this.ownerDocument.createElementFragment
()
        while(childNodes.length){
            fragment.appendChild(childNodes[0])
        }
        if(!!deep){
            parent.removeChild(this)
        }else{
            parent.replaceChild(this,fragment)
        }
        return this
    }
    //可以模拟wrap、wrapInner的效果
    HTMLElement.prototype.applyElement = function(newNode, wher
e) {
        newNode = newNode.removeNode(false)
        var range = this.ownerDocument.createRange()
        var where = ((where || 'outside').toLowerCase()
        var method = where === 'inside' ? 'selectNodeContents'
:
           where === 'outside' ? 'selectNode' : 'error'
        if(method === 'error'){
            throw new Error('DOMException.NOT_SUPPORTED_ERR(9)'
);
        }else{
            range[method](this)
            range.surroundContents(newNode)
            range.detach()
        }
        return newNode
    }
}
```

6.3　节点的复制

IE对元素的复制与innerHTML一样，存在许多bug，非常著名的就是上节所说的，IE自作多情地复制attachEvent事件。另外，根据标准浏览器cloneNode的测试，它只会复制元素写在标签内的属性与通过setAttribute设置的属性，而IE6～IE8还支持通过`node.aaa = "xxx"`设置的属性复制。

```
<div id="aaa" data-test="test" title="title">目标节点</div>

window.onload = function() {
    var node = document.getElementById("aaa");
    node.expando = {
        key: 1
    }
    node.setAttribute("attr", "attr")
    var clone = node.cloneNode(false);
    alert(clone.id);//aaa
    alert(clone.getAttribute("data-test"));//test
    alert(clone.getAttribute("title"));//title
    alert(clone.getAttribute("attr"));//attr
    node.expando.key = 2 //修正为2
    alert(clone.expando.key )//IE6～IE8：2；其他：undefined
}
```

如果仅是这样还好办，但IE在复制时不但会多复制一些，有时还会少复制一些，这让程序员不好处理。我们看一下jQuery是怎么处理的。jQuery.fn.clone拥有两个参数，第一个是只复制节点，但不复制数据与事件，默认为false；第二个决定如何复制它的子孙，默认是遵循参数一的决定。

```
jQuery.fn.clone = function( dataAndEvents, deepDataAndEvents )
{
    //方法只是用来调整这两个参数，然后交给真正干事的jQuery.clone
    dataAndEvents = dataAndEvents == null ? false : dataAndEven
ts;
    deepDataAndEvents = deepDataAndEvents == null ? dataAndEven
ts : deepDataAndEvents;
    return this.map( function() {
       return jQuery.clone(this,dataAndEvents,deepDataAndEvents
);
    });
},
```

jQuery.clone异常复杂。

```
jQuery.clone = function( elem, dataAndEvents, deepDataAndEvents
 ) {
       var destElements, node, clone, i, srcElements,
           inPage = jQuery.contains( elem.ownerDocument, elem )
;
       //判定浏览器对W3C规范是否支持足够良好，现在是chrome这样，可以直接
       //使用cloneNode(true)搞定
       if ( support.html5Clone || jQuery.isXMLDoc( elem ) ||
```

```
            !rnoshimcache.test( "<" + elem.nodeName + ">" ) ) {

            clone = elem.cloneNode( true );
        } else {
            //IE8不支持复制未知的标签类型，需要使用outerHTML hack
            fragmentDiv.innerHTML = elem.outerHTML;
            fragmentDiv.removeChild( clone = fragmentDiv.firstCh
ild );
        }

        if ( ( !support.noCloneEvent || !support.noCloneChecked
) &&
               ( elem.nodeType === 1 || elem.nodeType === 11 )
               && !jQuery.isXMLDoc( elem ) ) {
            //IE6～IE8下是使用attachEvent添加事件的，这时可能将事件也复
制了
            //此外如果元素是input[type=radio]，其checked属性可能无法复
制
            //这里有一大箩筐bug要修复
            destElements = getAll( clone );
            srcElements = getAll( elem );

            // 这里几乎都是IE6～IE8的bug
            for ( i = 0; ( node = srcElements[ i ] ) != null; ++
i ) {

                 // Ensure that the destination node is not null
; Fixes #9587
                 if ( destElements[ i ] ) {
                     fixCloneNodeIssues( node, destElements[ i ]
 );
                 }
            }
         }

         // 这是针对jQuery的行为，复制之前的数据与事件
         if ( dataAndEvents ) {
             if ( deepDataAndEvents ) {
                 srcElements = srcElements || getAll( elem );
                 destElements = destElements || getAll( clone );

                 for ( i = 0; ( node = srcElements[ i ] ) != nul
l; i++ ) {
                     cloneCopyEvent( node, destElements[ i ] );
                 }
```

```
            } else {
                cloneCopyEvent( elem, clone );
            }
        }

        // 复制生成的script节点与innerHTML一样，不会执行脚本或发出请求
        // 如果手动修复
        destElements = getAll( clone, "script" );
        if ( destElements.length > 0 ) {
            setGlobalEval( destElements, !inPage && getAll( ele
m, "script" ) );
        }

        destElements = srcElements = node = null;

        return clone;
}
```

getAll方法是用于获取此节点下所有子孙节点的，类似getElementsByTagName，不过它会根据浏览器是否支持querySelectorAll来判定是否进行优化。

```
function fixCloneNodeIssues( src, dest ) {
    var nodeName, e, data;

    // 此方法中处理元素节点
    if ( dest.nodeType !== 1 ) {
        return;
    }

    nodeName = dest.nodeName.toLowerCase();

    // 以前原来节点的事件
    if ( !support.noCloneEvent && dest[ jQuery.expando ] ) {
```

```
        data = jQuery._data( dest );

        for ( e in data.events ) {
            jQuery.removeEvent( dest, e, data.handle );
        }

        //移除UUID
        dest.removeAttribute( jQuery.expando );
    }

    // 手动添加script的text属性，IE下通过cloneNode无法复制
    if ( nodeName === "script" && dest.text !== src.text ) {
        disableScript( dest ).text = src.text;
        restoreScript( dest );

    // IE6～IE10 improperly clones children of object elements u
sing classid.
    // IE6～IE10无法复制object元素的孩子,新节点的下面没有孩子！！
    } else if ( nodeName === "object" ) {
        if ( dest.parentNode ) {
            dest.outerHTML = src.outerHTML;
        }
        if ( support.html5Clone && ( src.innerHTML && !jQuery.t
rim( dest.innerHTML ) ) ) {
            dest.innerHTML = src.innerHTML;
        }

    } else if ( nodeName === "input" && rcheckableType.test( sr
c.type ) ) {

        // IE6～IE8 无法复制checkbox/radio的checked, defaultCheck
ed属性
        dest.defaultChecked = dest.checked = src.checked;

        // IE6～IE7 checkbox/radio标签如果没有显示指定value时，其默认
值会变成on
        if ( dest.value !== src.value ) {
            dest.value = src.value;
        }

    // IE6～IE8 无法复制option元素的selected defaultSelected属性
    } else if ( nodeName === "option" ) {
        dest.defaultSelected = dest.selected = src.defaultSelec
ted;
```

```
    // IE6～IE8 无法复制文本域、文本区的defaultValue属性
    } else if ( nodeName === "input" || nodeName === "textarea"
 ) {
        dest.defaultValue = src.defaultValue;
    }
}
```

上面处理object标签，其实还是有问题的。大家要自己实现库时，建议改成如下。

```
if(nodeName === 'object'){
   var params = src.childNodes
   if(dest.childNodes.length !== params.length){
      for(var i = 0, el; el = params[i++];){
         dest.appendChild(el.cloneNode(true))
      }
   }
}
```

至于cloneCopyEvent，这方法与jQuery的数据缓存系统绑得太死了，暂时先跳过。

不过随着浏览器的升级，原生的cloneNode方法的bug会修复得很快，因此大家不用支持IE6～IE8这样古老的浏览器，可以像zepto那样用很有魄力的几行搞定：

```
//zepto
clone: function() {
    return this.map(function() {
        return this.cloneNode(true)
    })
}
```

6.4 节点的移除

浏览器提供了多种移除节点的方法，常见的有removeChild、removeNode，动态创建一个元素节点或文档碎片再appendChild，创建Range对象选中目标节点然后deleteContents。

```
//一个不需要知道父节点，移除节点的方法
var f = document.createDocumentFragment()
function clearChild (node) {
     f.appendChild(node)
     f.removeChild(node)
     return node
 }
```

removeNode是IE的私有实现，Opera也实现了此方法。它的作用是将目标节点从文档树中删除，返回目标节点。它有一个参数，为布尔值，其默认值为false，即仅删除目标节点，保留子节点，为true时相当于removeChild。

deleteContents算是比较偏门的API，兼容性

差，这个以后说。

removeChild在IE6～IE7中有内存泄漏问题，由IE的GC回收比较失败而引起。由于这太底层了，这里就不展开了。这里给出EXT框架的方案。像EXT这样庞大的UI库，所有节点都动态生成，因此是非常注重GC回收的。

```
var removeNode = IE6 || IE7 ? function() {
    var d; //IE6、IE7的判定自己写
    return function(node) {
        if (node && node.tagName != 'BODY') {
            d = d || document.createElement('DIV');
            d.appendChild(node);
            d.innerHTML = '';
        }
    }
}() : function(node) {
    if (node && node.parentNode && node.tagName != 'BODY') {
        node.parentNode.removeChild(node);
    }
}
```

为什么这么写呢？因为在IE6～IE8中存在一个叫**DOM** 超空间（DOM hyperspace）的概念。即当元素移出DOM树，又有JavaScript关联时元素不会

消失，它被保存在这个叫超空间的地方。《PPK谈JavaScript》一书指出，可以用是否存在parentNode来判定元素是否在超空间。

```
window.onload = function() {
    var div = document.createElement("div");
    alert(div.parentNode); //null
    document.body.removeChild(document.body.appendChild(div));
    alert(div.parentNode); //IE6～IE8 object;其他 null
    if (div.parentNode) {
        alert(div.parentNode.nodeType); //11 文档碎片
    }
}
```

第一个alert出null，这个所有浏览器都一样，因此有时我们误以为可以当作节点是否在DOM的基准。但当元素插入DOM树再移出时，就有差异了，IE6～IE8会下弹出一个文档碎片对象。因此可以想象为何IE性能这么差了，它自以为这样能重复使用元素，但通常用户移除了就不管，因此久而久之，内存就允许了许多这样的“碎片”，加之其他问题，就很容易造成泄漏。

我们再看innerHTML清除元素会怎么样。

```
<body><div id="test"></div></body>

window.onload = function() {
    var div = document.getElementById('test');
    document.body.innerHTML = '';
    alert(div.parentNode);//null
}
```

结果在IE下也是null，但这也不能说明innerHTML就比removeChild好。我们继续下一个实验。

```
<body>
    <div><div id="test1">test1</div></div>
    <div><div id="test2">test2</div></div>
</body>


window.onload = function() {
    var div1 = document.getElementById('test1');
    div1.parentNode.removeChild(div1);
    alert(div1.id + ":" + div1.innerHTML);//test1:test1
    var div2 = document.getElementById('test2');
    div2.parentNode.innerHTML = "";
    alert(div2.id + ":" + div2.innerHTML);//test2
}
```

这时我们就发现，当用removeChild移除节点，

原来元素的结构没有发生变化，但用innerHTML时，IE6～IE8下会直接清空其里面的内容，只剩下个空壳，而标准浏览器则与removeChild保持一致。打个比喻，在IE下，removeChild就是掰断树枝，但树枝可以再次使用。而innerHTML就是把所需要的枝叶给拔下来然后把树枝烧掉。鉴于 IE 对内存管理的失败，这么干净的清除节点正是我们寻找的方法！因此EXT从1.0到4.0，此方法也没大改变。

对于jQuery这样的类库框架来说，估计很难走这条路。它已经被自己的数据缓存系统绑架了，移除节点时需要逐个检测元素，从缓存系统中移除对应的缓存体，否则会让浏览器宕机。不过最不好的是jQuery通过类数组结构与 preObject困住节点的方式，这就造成了jQuery即便是使用innerHTML，元素节点在IE下还是位于DOM超空间中。

jQuery 在性能上没有优势，于是在移除节点的

方式上造势。它提供了 3 种移除节点的方式：remove，移除节点的同时从数据缓存系统上移除对应数据；empty，只清空元素的内部，相当于IE下的removeNode(false)；及detach方法。经常有这么一个场景，我们要为元素做一些复杂的属性或样子操作，出于性能考虑，我们会先将元素移出DOM树，待处理完再插回来。但绝对大多数操作DOM的方法都与数据缓存方法关联在一起，若用remove方法，会将这些数据与事件都清理掉的。因此，**detach** 方法 应运而生，它只是用于临时移出DOM树，不会移除数据与事件。

下面是它们的实现。

```
"remove,empty,detach".replace(/[^, ]+/g, function(method) {
    $.fn[method] = function() {
        var isRemove = method !== "empty";
        for (var i = 0, node; node = this[i++];) {
            if (node.nodeType === 1) {
                //移除匹配元素
                var array = $.slice(node[TAGS]("*")).concat(isR
emove ? node : []);
                if (method !== "detach") {
```

```
                    array.forEach(cleanNode);
                }
            }
            if (isRemove) {
                if (node.parentNode) {
                    node.parentNode.removeChild(node);
                }
            } else {
                while (node.firstChild) {
                    node.removeChild(node.firstChild);
                }
            }
        }
        return this;
    }
});
```

如果我们的框架没有像jQuery那样引入一个庞大的数据缓存系统，而是像zepto.js那样通过HTML5的data-*来缓存数据，那么许多东西都可以简化了。这也意味着我们不打算兼容IE6、IE7、IE8，那么我们就可以使用deleteContents或textContent。比如我们实现一个清空元素内部的API。

版本一，最传统的方式。

```
function clearChild (node) {//node可以是元素节点与文档碎片
        while (node.firstChild) {
```

```
            node.removeChild(node.firstChild)
        }
        return node
 }
```

版本二，使用deleteContents，创建一个Range对象，然后通过setStartBefore、setEndAfter选择边界，最后清空它们俩的节点。

```
var deleteRange = document.createRange()
 function clearChild (node) {//node可以是元素节点与文档碎片
      deleteRange.setStartBefore(node.firstChild)
      deleteRange.setEndAfter(node.lastChild)
      deleteRange.deleteContents()
      return node
 }
```

版本三，使用textContent。textContent是W3C版本的innerText。这个东西在较新的浏览器中兼容性特别好，并且同时存在于元素节点与文档碎片中。

```
function clearChild (node) {//node可以是元素节点与文档碎片
     node.textContent = ""
     return node
 }
```

6.5　节点的移除回调实现

当我们为元素节点绑定了许多数据与事件时，为了防止内存泄漏，有时我们需要一个回调，当节点被移除时执行一些清理操作。

这是一个很高级的抽象，早期浏览器是没有对应API的。到IE9时，出现了很短命的Mutation Event，其中与移除相关的事件有两个。

- DOMNodeRemoved，如果节点被其包含的父节点移除，就会触发此事件。
- DOMNodeRemovedFromDocument，如果节点被其包含的父节点或其祖先节点移除，就会触发此事件。

从功效来看，显然**DOMNodeRemovedFromDocument** 更加好用

些，因为我们不知道哪一级父节点会发生removeChild或innerHTML操作，导致下方所有子孙都被清空。

```
if(window.chrome){//现在只有chrome坚持支持此事件
    var root = document.documentElement
    root.addEventListener('DOMNodeRemovedFromDocument', functio
n(e){
      setTimeout(function(){//判定这是永久移除还是临时的节点挪动
          if(root.contains(e.target)){//如果还在DOM树，说明是永久移
除
             action() //这里执行你的清理操作
          }
      })
    })
}
```

但随着Mutation Event的昙花一现，我们需要寻找新的替代品。浏览器将监听节点变动的工作交给Mutation Observer。

6.5.1　Mutation Observer

Mutation Observer（变动观察器）是监视DOM变动的接口。DOM发生任何变动，Mutation

Observer都会得到通知。

概念上，它很接近事件，可以理解为，当DOM发生变动，会触发Mutation Observer事件。但是，它与事件有一个本质不同：事件是同步触发，也就是说，当DOM发生变动，立刻会触发相应的事件；Mutation Observer则是异步触发，DOM发生变动以后，并不会马上触发，而是要等到当前所有DOM操作都结束后才触发。

这样设计是为了应付DOM变动频繁的特点。举例来说，如果在文档中连续插入1000个段落（p元素），就会连续触发1000个插入事件，执行每个事件的回调函数，这很可能造成浏览器的卡顿；而Mutation Observer完全不同，只在1000个段落都插入结束后才会触发，而且只触发一次。

Mutation Observer有以下特点。

（1）它等待所有脚本任务完成后，才会运行，即采用异步方式。

（2）它把DOM变动记录封装成一个数组进行处理，而不是一条条地个别处理DOM变动。

（3）它既可以观察发生在DOM的所有类型变动，也可以观察某一类变动。

目前，Firefox（14+）、Chrome(26+)、Opera（15+）、IE（11+）和Safari（6.1+）支持这个API。Safari 6.0和Chrome 18～Chrome 25使用这个API的时候，需要加上WebKit前缀（WebKitMutationObserver）。可以使用下面的表达式，检查当前浏览器是否支持这个API。

```
var MutationObserver = window.MutationObserver
  || window.WebKitMutationObserver
  || window.MozMutationObserver
var observeMutationSupport = !!MutationObserver
```

MutationObserver是一个构造函数，需要传入一个回调作参数，返回一个observer对象，然后observer需要指定另外两个参数（第一个是监听的起点；第二个是配置对象，指明要监听哪些类型的变动）

```
var observer = new MutationObserver(callback);
var  options = {
  'childList': true,
  'attributes':true
} ;
observer.observe(document.documentElement, options);
```

options配置对象拥有如下属性。

- childList：布尔，子节点的变动。
- attributes：布尔，属性的变动。
- characterData：布尔，节点内容或节点文本的变动。
- subtree：布尔，所有后代节点的变动。
- attributeOldValue：布尔，表示观察attributes变

动时，是否需要记录变动前的属性值。

- characterDataOldValue：布尔，表示观察characterData变动时，是否需要记录变动前的值。
- attributeFilter：数组，表示需要观察的特定属性（比如[‘class’,’src’]）。

此外还有其他API或方法，具体可看MDN。目前的知识已经够我们实现一个新的remove监控方法了！

```
https://developer.mozilla.org/zh-CN/docs/Web/API/MutationObserv
er
//inspired by http://stackoverflow.com/questions/31798816/simpl
e-mutationobserver-version- of-domnoderemovedfromdocument
function onRemove(element, onDetachCallback) {
    var observer = new MutationObserver(function () {
        function isDetached(el) {
            if (el.parentNode === document) {
                return false
            } else if (el.parentNode === null) {
                return true
            } else {
                return isDetached(el.parentNode)
            }
        }
        if (isDetached(element)) {
            onDetachCallback()
        }
    })
```

```
    observer.observe(document, {
        childList: true,
        subtree: true
    })
}
```

6.5.2　更多候选方案

但无论是Mutation Event还是MutationObserver，对浏览器的版本要求还是非常高的，因此我们需要更多候选方案。在avalon中，为了实现对组件的生命周期监控，在移除节点这方面一共提供了4个方法。

方案一，如果浏览器是自定义标签，这里特指通过document.register方法注册的unresolved元素，否则它们就是未知元素，没有生命周期钩子，如表6-3所示。

表**6-3**

类型	继承自	示例

unresolved元素	HTMLElement	<x-tabs>、<my-element>、<my-awesome-app>
未知元素	HTMLUnknownElement	<tabs>、<foo_bar>

目前，Chrome（27+）和Firefox（23+）都提供了对`document.register()`的支持，不过之后规范又有一些演化。chrome 31将是第一个支持新规范的版本。

创建一个自定义标签如下。

```
var XFoo = document.register('x-foo', {
  prototype: Object.create(HTMLElement.prototype, {
    createdCallback:funciton(){ alert("节点创建时的回调") },
    bar: {
      get: function() { return 5; }
    },
    foo: {
      value: function() {
        alert('foo() called');
      }
    }
  })
});
```

这个新标签的原型中可以添加4个生命周期钩

子，如表6-4所示。

表**6-4**

回调名称	调用时间点
createdCallback	创建元素实例
attachedCallback	向文档插入实例
detachedCallback	从文档中移除实例
attributeChangedCallback	添加，移除，或修改一个属性

基于此，我们得到第一个实现方案。

```
//https://github.com/RubyLouvre/avalon/blob/master/src/componen
t/disposeDetectStrategy.js
//用于chrome, safari
var tags = {}
function byCustomElement(name) {
    if (tags[name])
        return
    var prototype = Object.create(HTMLElement.prototype)
    tags[name] = prototype
    prototype.detachedCallback = function () {
        var dom = this
        setTimeout(function () {
            fireDisposeHook(dom)
        })
    }
    document.registerElement(name, prototype)
```

```
}
```

方案二，使用Mutation Event。

```
function byMutationEvent(dom) {
    dom.addEventListener("DOMNodeRemovedFromDocument", function
 () {
        setTimeout(function () {
            fireDisposeHook(dom)
        })
    })
}
```

方案三，如果是IE9或以上，window上暴露了Node构造器，它是HTMLElement更上位的原型对象，通过改写一些常用的DOM操作方法，我们就能得到移除的时机了。

```
function byRewritePrototype() {
    if (byRewritePrototype.execute) {
        return
    }

    byRewritePrototype.execute = true
    var p = Node.prototype
    function rewite(name, fn) {
        var cb = p[name]
        p[name] = function (a, b) {
            return  fn.call(this, cb, a, b)
        }
```

```
}
rewite('removeChild', function (fn, a, b) {
    fn.call(this, a, b)
    if (a.nodeType === 1) {
        setTimeout(function () {
            fireDisposeHook(a)
        })
    }
    return a
})

rewite('replaceChild', function (fn, a, b) {
    fn.call(this, a, b)
    if (b.nodeType === 1) {
        setTimeout(function () {
            fireDisposeHook(a)
        })
    }
    return a
})

rewite('innerHTML', function (fn, html) {
    var all = this.getElementsByTagName('*')
    fn.call(this, html)
    fireDisposedComponents(all)
})

rewite('appendChild', function (fn, a) {
    fn.call(this, a)
    if (a.nodeType === 1 && this.nodeType === 11) {
        setTimeout(function () {
            fireDisposeHook(a)
        })
    }
    return a
})

rewite('insertBefore', function (fn, a) {
    fn.call(this, a)
    if (a.nodeType === 1 && this.nodeType === 11) {
        setTimeout(function () {
            fireDisposeHook(a)
        })
    }
    return a
```

```
    })
}
function fireDisposedComponents (nodes) {
    for (var i = 0, el; el = nodes[i++]; ) {
         fireDisposeHook(el)
    }
}
```

方案四，最简单粗暴低效的轮询，用于应付IE6～IE8。

```
var checkDisposeNodes = []
var checkID = 0
function byPolling(dom) {
    avalon.Array.ensure(checkDisposeNodes, dom)
    if (!checkID) {
        checkID = setInterval(function () {
            for (var i = 0, el; el = checkDisposeNodes[i++]; )
{
                if (false === fireDisposeHook(el)) {
                    avalon.Array.removeAt(checkDisposeNodes, i)

                    --i
                }
            }
            if (checkDisposeNodes.length == 0) {
                clearInterval(checkID)
                checkID = 0
            }
        }, 1000)
    }
}
```

至于为什么不用MutationObserver，因为在IE

下，如果使用此方法，那么它原本是单个文本节点的地方（暂时发现在{{}}会断成N个文本节点，称之为碎片化。比如说，明明这个元素节点下只有3个文本节点，结果多出11个，由于文本节点的内容被拆得支离破碎，就很可能会导致一些基于DOM的MVVM框架扫描失败。

```
https://github.com/RubyLouvre/avalon/issues/636
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=Edge" />
        <title>TEST</title>

        <script>

            var MutationObserver = window.MutationObserver
            if (MutationObserver) {
                var observer = new MutationObserver(function(mu
tations) {
                    //...
                });

                observer.observe(document.documentElement, {
                    childList: true,
                    subtree: true
                })

            }
            window.onload = function() {
                setTimeout(function() {
                    var el = document.getElementById("aaa")
                    console.log(el.childNodes.length)
                }, 3000)
```

```
            }

        </script>

    </head>
    <body>
        <div id="aaa">我的{{name}}叫{{answer}},他的{{name}}叫{{no
}},{{10*10}}
            <p>IE下，如果使用了MutationObserver，那么它原本是单个文本
节点的地方（暂时发现在{{）会断成<em>N</em>个文本节点(我称之为碎片化)，导
致一些基于DOM的MVVM框架扫描失败</p>
        </div>
    </body>
</html>
```

6.6 innerHTML、innerText、outerHTML、outerText的兼容处理

IE率先实现了innerHTML、innerText、outerHTML、outerText这4个便捷的API来替换元素节点的内容或本身，如图6-3所示。

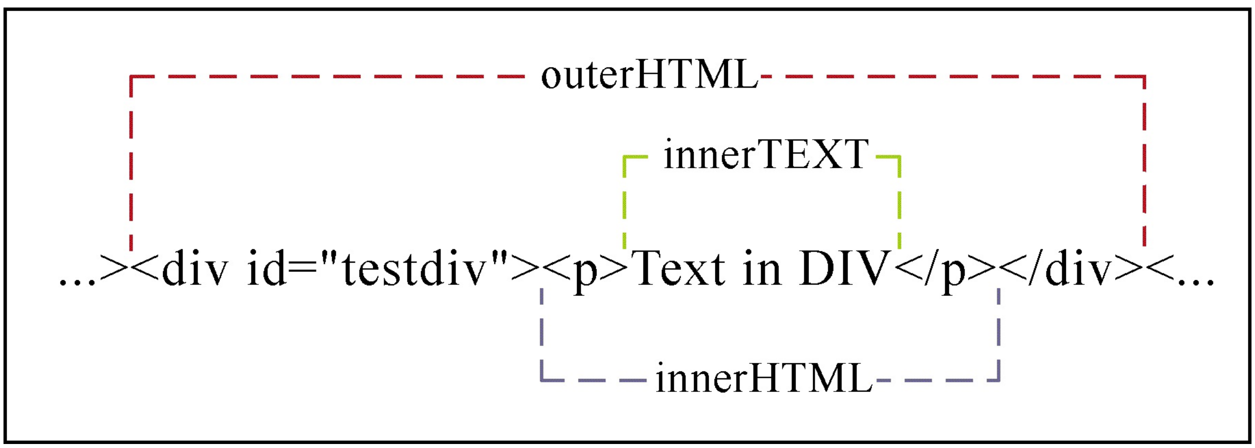


图6-3

outerText的作用区域与outerText是一样的。

功能如下。

- innerHTML，设置或获取位于元素开标签与闭标签之间的HTML。
- outerHTML，设置或获取元素及其内容的HTML。
- innerText，设置或获取位于元素开标签与闭标签之间的文本。
- outerText，设置或获取元素及其内容的文本。

其中由于innerHTML能批量生成节点，很快就普及到其他浏览器。目前这些方法都被W3C标准化，只有Firefox的实现比较迟缓，如图6-4～图6-8所示。

> Firefox39：



图6-4

IE11：



图6-5

IE11的IE6模式：



图6-6

Chrome49：

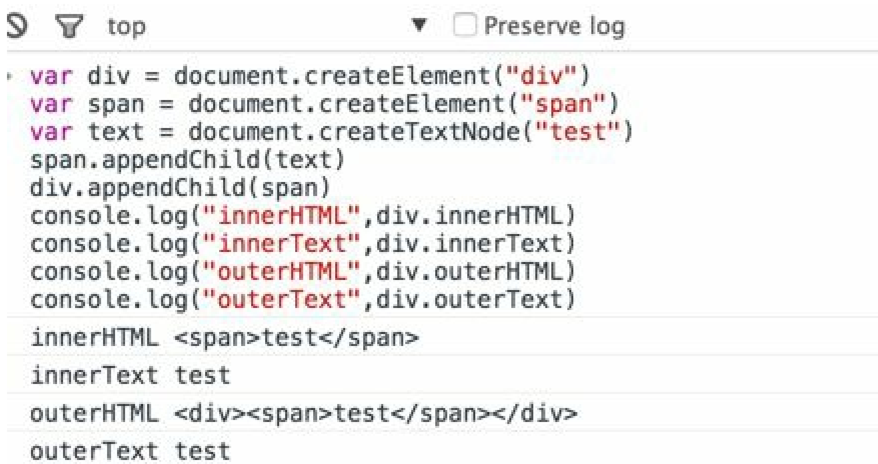


图6-7

Safari9：

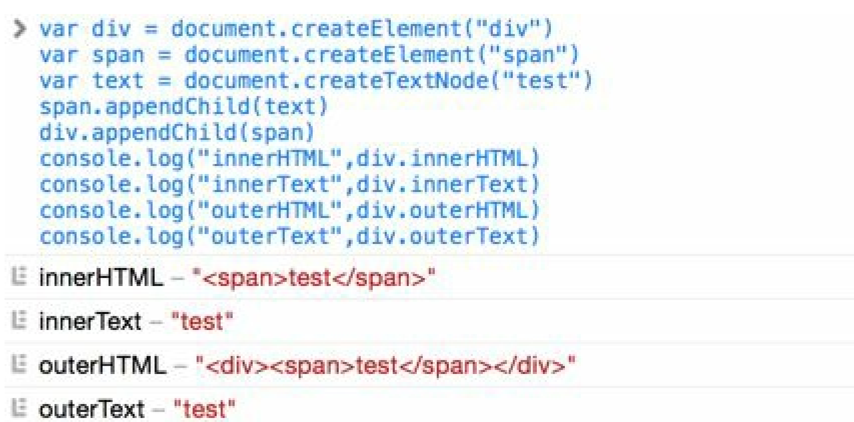


图6-8

我们首先从简单的innerText啃起。虽然标准浏

览器有一个很相似的textContent来实现此功能，但它与innerText又有稍微的区别。

（1）textContent会获取所有元素的content，包括<script> 和<style> 元素。

（2）innerText不会获取display:none的元素的content，而textContent会。

（3）innerText会触发reflow，而textContent不会。

（4）innerText返回值会被格式化，而textContent不会。

在firefox下兼容它们，基本上是使用Range对象搞定：

```
var p = typeof HTMLElement !== 'undefine' && HTMLElement.prototype
if (!('outerHTML' in p)) {
    p.__defineSetter__('outerHTML', function (s) {
        var r = this.ownerDocument.createRange()
```

```
        r.setStartBefore(this)
        var df = r.createContextualFragment(s)
        this.parentNode.replaceChild(df, this)
        return s
    });
    p.__defineGetter__('outerHTML', function () {
        var a = this.attributes, str = '<' + this.tagName, i =
0;
        for (; i < a.length; i++) {
            if (a[i].specified) {
                str += ' ' + a[i].name + '=' + JSON.stringify(a
[i].value)
            }
        }
        if (!this.canHaveChildren)
            return str + ' />'
        return str + '>' + this.innerHTML + '</' + this.tagName
 + '>'
    })

    p.__defineGetter__('canHaveChildren', function () {
        return !/^(area|base|basefont|col|frame|hr|img|br|input
|isindex|link|meta|param)$/
        i.test(this.tagName)
    })
}

if (!('innerText' in p)) {
    p.__defineSetter__('innerText', function (sText) {
        var parsedText = document.createTextNode(sText)
        this.innerHTML = ""
        this.appendChild(parsedText)
        return parsedText
    })
    p.__defineGetter__('innerText', function () {
        var r = this.ownerDocument.createRange()
        r.selectNodeContents(this)
        return r.toString()
    })

}

if (!('outerText' in p)) {
    p.__defineSetter__("outerText", function (sText) {
        var parsedText = document.createTextNode(sText)
```

```
        this.parentNode.replaceChild(parsedText, this)
        return parsedText
    })
    p.__defineGetter__('outerText', function () {
        var r = this.ownerDocument.createRange()
        r.selectNodeContents(this)
        return r.toString()
    })
}
```

如果想兼容XML，还可以使用以下方法：

```
function outerHTML(el) { //主要是用于XML
     switch (el.nodeType + '') {
         case '1':
         case '9':
             return "xml" in el ? el.xml : new XMLSerializer().
serializeToString(el)
         case '3':
             return el.nodeValue
         case '8':
             return '<!--'+ el.nodeValue+'-->'
         default:
             return ''
     }
}

function innerHTML(el) { //主要是用于XML
     for (var i = 0, c, ret = []; c = el.childNodes[i++];) {
         ret.push(outerHTML(c))
     }
     return ret.join("")
}

function getText() {
     //获取某个节点的文本，如果此节点为元素节点，则取其childNodes的所有文本
     return function getText(nodes) {
         for (var i = 0, ret = '' node; node = nodes[i++];) {
           // 处理的文本节点与CDATA的内容
```

```
                if (node.nodeType === 3 || node.nodeType === 4) {

                    ret += node.nodeValue     //取得元素节点的内容
                } else if (node.nodeType !== 8) {
                    ret += getText(node.childNodes)
            }
            }
            return ret
        }
}()
```

6.7　模板容器元素

随着前端模板或MVVM的流行，我们需要一些元素作为容器或作为模板的载体。最常见的是，使用script/textarea来存放模板代码，然后使用innerHTML/value属性来获取模板内容进行解析和拼装。

```
<script type="text/x-template" id="tpl">
  <h1><%=data.title%></h1>
  <p><%=data.content%></p>
</script>
<script>
  var htmlTpl = document.getElementById("tpl").innerHTML;
  tplEngine(htmlTpl, {
    title: "This is title",
    content: "This is content"
  });
</script>
```

本章节主要比较现有模板容器元素的优缺点，及介绍HTML5新出品的template元素。

`<script>`，我们通过取其text属性，并改写其

type属性为一个浏览器不认识的MIME值，它的内容就不会执行。它的优势是天然不显示。缺点是不能套嵌出现script标签。

<noscript>，在浏览器普遍支持JavaScript的情况下，作为模板容器算是为它找到一份副业。但它有严重的兼容性问题，因此建议只用于IE9＋。具体表现为IE7、IE8使用innerText、innerHTML都无法得其内容，IE9只能用innerText。下面是avalon1.4时的兼容方法：

```
var rnoscript = /<noscript.*?>(?:[\s\S]+?)<\/noscript>/img
function getNoscriptText(el) {
    //IE9～IE11与Chrome的innerHTML会得到转义的内容，它们的innerText可以
    if (el.textContent && /\S+/.test(el.textContent)) {
        return el.textContent
    }
    //IE7～IE8 innerText,innerHTML都无法取得其内容，IE6能取得其innerHTML
    if (IEVersion === 6 || IEVersion > 8 || window.netscape) {
        return el.innerHTML
    }
    //IE7、IE8需要用AJAX请求得到当前页面进行抽取
    var xhr = getXHR()
    xhr.open("GET", location, false)
    xhr.send(null)
    //http://bbs.csdn.net/topics/390349046?page=1#post-393492653
    var noscripts = DOC.getElementsByTagName("noscript")
```

```
    var array = (xhr.responseText || "").match(rnoscripts) || [
]
    var n = array.length
    for (var i = 0; i < n; i++) {
        var tag = noscripts[i]
        if (tag) {
            //IE6～IE8中noscript标签的innerHTML,innerText是只读的
            //http://blog.jobbole.com/50787/
            tag.style.display = "none"
            tag.textContext = (array[i].match(rnoscriptText) ||
 ["", " "])[1]
        }
    }
    return el.textContent
}
```

<textarea>，这个也很常用，但有两个缺点，一是天然不隐形；二是需要额外的设置display: none，并且内容不能再套<textarea>。

<xmp>，这是一个很老很特殊的标签，语义为example，示例。据说后来被<pre>标签取代而废止，实际上，目前所有的浏览器都是支持的。但是，其跟<pre>标签不能划等号。<pre>里面有个<img>标签，它会转换为一个元素节点作为孩子。而<xmp>里面无法加多个标签，里面只有一个文本

节点作为独子。由于是文本节点，因此里面的img、script标签不会发出请求，也不存在套嵌问题。`<xmp>` 是旧时代作为模板容器的最佳选择，缺点只有一个，天然不隐形。

`<template>` 最早是Chrome为了支持polymer项目，自己实现的一个新元素。相比上面那些元素，拥有以下优势。

（1）元素本身是天然隐形，不需要设置display: none。

（2）元素可以放在任意位置。

（3）比xmp更加激进，它里面一个节点也没有。但通过访问content属性，你能得到其文档碎片对象；访问innerHTML，可以得到之前的用户设置的内容。由于没有节点，这样就不影响querySelectorAll的效率。

（4）template元素可以继续套template元素。里面的img、script节点不会发出请求，里面的style元素不会影响外面的样式。

唯一不济的是，它的兼容性不够好，如图6-9所示。

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	26	22 (22)	Edge 13	15	7.1

图6-9

6.8　iframe元素

iframe是一个古老的标签，在IE3时已经存在了。由于它是用作镶嵌另一个页面到主页面，因此肯定与一般的元素不同，创建起来也不是一般的消耗资源，并且消耗连接数。但它却是一个物超所值的东西，有了它，我们可以无障碍地实现无缝刷新，通过保存历史模拟onhashchange，安全地加载第三方资源与广告，实现富文本编辑器、文件上传，用它搞定IE6～IE7的select bug，在iframe里做特征侦测……HTML5为它添加了3个属性，让它变得更强大。

由于iframe太古老，因此相关的兼容性问题及对应的hack不在少数，本节就特意详细地讲解它。

首先是样式问题。

（1）想要隐藏iframe那个很粗的边框时，我们需要用到frameBorder属性，例如使用Dreamweaver可以生成如下代码。

```
<iframe frameborder=0 src='xxxx' width='xxx' height='xxx'></ifr
ame>
```

换成JavaScript，需要这样写：

```
var iframe = document.createElement('iframe');
iframe.frameBorder=0;//Firefox和IE均有效，相当于CSS中的"border:0 no
ne"
```

（2）去掉iframe中的滚动条：

```
iframe.scrolling = "no";
```

（3）设置iframe透明，需要分两步走。

① iframe自身设置allowTransparency属性为true（但设置了allowTransparency=true就遮不住

select）。

② iframe中的文档，background-color或body元素的bgColor属性必须设置为transparent。

具体代码如下。

```
// 1. 包含iframe页面的代码。
<body bgColor="#eeeeee"> <iframe allowTransparency="true" src="
transparent.htm"> </iframe>
// 2. transparent.htm页的代码。
<body bgColor="transparent">
```

然后是获取iframe中的各种重要对象。

（1）获取iframe中的window对象。

```
function getIframeWindow(node) {
    return node.contentWindow;
}
```

（2）获取iframe中的文档对象。

```
function getIframeDocument(node) { //w3c || IE
    return node.contentDocument || node.contentWindow.document;
```

```
}
```

判定页面是否在iframe里面。

```
window.onload = function() {
    alert(window != window.top)
    alert(window.frameElement !== null);
    alert(window.eval !== top.eval)
}
```

判定iframe是否加载完毕。

```
if (iframe.addEventListener) {
    iframe.addEventListener("load", callback, false);
} else {
    iframe.attachEvent("onload", callback)
}
```

不过，如果是动态创建iframe时，webkit系统浏览器可能会出现二次触发onload事件的问题。

```
<div id="times">0</div>
<script>
    window.onload = function(){
      var c = document.getElementById("times");
      var iframe = document.createElement("iframe");
      iframe.onload = function(){ c.innerHTML = ++c.innerHTML }
```

```
      document.body.appendChild(iframe);
      iframe.src = "http://www.cnblogs.com/rubylouvre"
    }
</script>
```

估计Safari和Chrome在appendChild之后就进行第一次加载，并且在设置src之前加载完毕，所以触发了两次。如果在插入body之前给iframe随便设置一个src（除了空值），间接加长第一次加载，那么也只触发了一次。不设置或空值的src相当于链接到“about:blank”（空白页）。

动态创建iframe时，如果想用到name属性，用document.createElement("iframe")创建再设置它的name属性，IE6、IE7是无法辨识此值的。

```
window.onload = function(){
    var iframe = document.createElement("iframe");
    iframe.name = "xxx";
    document.body.appendChild(iframe);
    iframe.src = "http://www.cnblogs.com/rubylouvre/"
    alert(frames["xxx"]);//undefined
     alert(document.getElementsByName("xxx").length)//0
}
```

因此我们需要针对IE6、IE7，使用IE特有的创建元素时连属性一起创建的方式实现。

```
if ("1"[0]) {//IE6、IE7这里返回undefined,于是跑到第二个分支
    var iframe = document.createElement("iframe");
    iframe.name = name;
} else {
    iframe = document.createElement('<iframe name="' + name + '
">');
}
```

iframe与父窗口共享history，基于它我们可以解决Ajax的后退按钮问题。里面的细节非常多，这里就不展开了，github上有两个著名的项目，可以帮你解决工作的绝大多数问题。

```
https://github.com/browserstate/history.js
https://github.com/devote/HTML5-History-API
```

清空iframe内容，不保留历史的写法：

```
iframeWindow.location.replace('about:blank');
```

IE6中iframe.src="about:blank"在https协议下会出现问题，需要用javascript:false修正，虽然速度非常慢。

6.9　总结

节点操作是驱动页面变动的核心，围绕它的API多如牛毛。本章挑出来的东西只是九牛一毛，想深入还需要经常浏览MSDN，MDN等网站。对不同浏览器的私有API的掌握程度，直接决定了你解决兼容性问题的思路与水准。大家在日常开发中一点点积累吧，如图6-10所示。

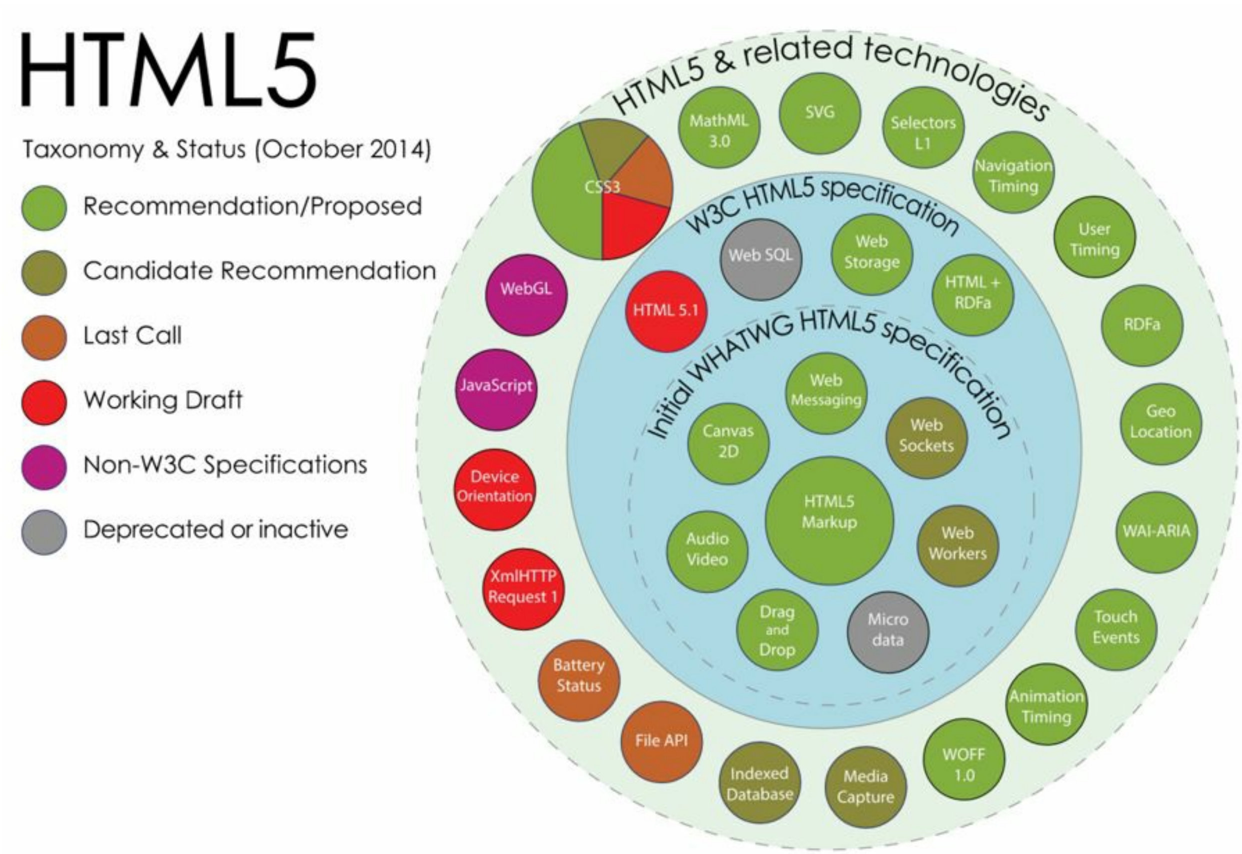


图6-10

HTML5带来一系列新API，让DOM编程更加丰富多彩，而本章乃致全书也不能涵盖这所有内容，因此大家需要时刻留意浏览器厂商的官网。

---

[1] DOM4中一共添加了before、after、replaceWith、remove、prepend、append 6个增删节点的API。

第7章　数据缓存模块



数据缓存系统最早应该是jQuery 1.2引入的，它是用于关联操作对象和与之相关的数据的一种机制。在DOM中，我们通常操作的数据有3种，元素

节点、文档对象与window对象。那时jQuery的事件系统完全照搬DE“大牛人”的addEvent.js，而addEvent在实现上有个缺憾，它把事件的回调都放到EventTarget之上，这会引发循环引用，如果EventTarget是window对象，又会引发全局污染。有了数据缓存系统，除了规避这两个风险外，我们还可以有效地保存不同方法产生的中间变量，而这些变量会对另一个模块的方法有用，解耦方法间的依赖。对于jQuery来说，它的事件克隆乃至后来的队列实现都离不开缓存系统。

7.1 jQuery的第1代缓存系统

jQuery1.2在core模块新增了两个静态方法，data与removeData。data不用说，与jQuery其他方法一样，读写结合。jQuery的缓存系统是把所有数据都放在`$.cache`仓库之上，然后为每个要使用缓存系统的元素节点、文档对象与window对象都分配一个UUID。UUID的属性名为一个随机的自定义属性，`"jQuery" + (new Date()).getTime()`，值为整数，从零递增。但UUID总要附于一个对象上，如果那个对象是window，岂不是全局污染吗？因此jQuery内部判定它是window对象时，映射为一个叫windowData的空对象，然后UUID加在它之上。有了UUID，我们在首次访问缓存系统时，会在`$.cache`对象开辟一个空对象（缓存体），用于放置与目标对象有关的东西。这有点像“银行开户”，UUID的值就是“存折”。removeData则会删掉

不再需要保存数据，如果到最后，数据删光了，它也没有任何键值对应，则成为空对象，jQuery就会从$.cache 中删掉此对象，并从目标对象移除UUID。

```
//jQuery1.2.3
var expando = "jQuery" + (new Date()).getTime(), uuid = 0, wind
owData = {};
jQuery.extend({
    cache: {},
    data: function( elem, name, data ) {
        elem = elem == window ? windowData :elem;
        //对window对象做特别处理
        var id = elem[ expando ];
        if ( !id ) //如果没有UUID则新设一个
            id = elem[ expando ] = ++uuid;
        //如果没有在$.cache中开户，则先开户
        if ( name && !jQuery.cache[ id ] )
            jQuery.cache[ id ] = {};

        // 第3个参数不为undefined时，为写操作
        if ( data != undefined )
            jQuery.cache[ id ][ name ] = data;
        //如果只有1个参数，则返回缓存对象，2个参数则返回目标数据
        return name ?  jQuery.cache[ id ][ name ] :   id;
    },

    removeData: function( elem, name ) {
        elem = elem == window ? windowData :   elem;
        var id = elem[ expando ];
        if ( name ) {//移除目标数据
            if ( jQuery.cache[ id ] ) {
                delete jQuery.cache[ id ][ name ];
                name = "";

                for ( name in jQuery.cache[ id ] )
                   break;
                //遍历缓存体，如果不为空，那name会被改写，如果没有被改写
```

```
，则!name 为true
                //从而引发再次调用此方法，但这次是只传一个参数，移除缓存体
                if ( !name )
                    jQuery.removeData( elem );
            }
        } else {
            //移除UUID，但IE下对元素使用delete会抛错
            try {
                delete elem[ expando ];
            } catch(e){
                if ( elem.removeAttribute )
                    elem.removeAttribute( expando );
            }//注销账户
            delete jQuery.cache[ id ];
        }
    }
})
```

jQuery在1.2.3版本中添加了两个同名的原型方法data与removeData，目的是方便链式操作与集化操作，并在data中添加了getData、setData的自定义事件的触发逻辑。

在jQuery 1.3中，数据缓存系统终于独立成一个模块data.js（内部开发时的划分），并添加了两组方法，分别是命名空间上的queue与dequeue和原型上的queue与dequeue。queue的目的很明显，就是

缓存一组数据，为动画模块服务。dequeue是从一组数据中删掉一个。

```
//jQuery1.3
jQuery.extend({
    queue: function( elem, type, data ) {
        if ( elem ){
            type = (type || "fx") + "queue";
            var q = jQuery.data( elem, type );
            if ( !q || jQuery.isArray(data) )//确保储存的是一个数组
                q = jQuery.data( elem, type, jQuery.makeArray(d
ata) );
            else if( data )//然后往这个数据加东西
                q.push( data );
        }
        return q;
    },
    dequeue: function( elem, type ){
        var queue = jQuery.queue( elem, type ),
        fn = queue.shift();//然后删掉一个，早期它是放置动画的回调，删
掉它就调用一下
        // 但没有做是否为函数的判定，估计也没有写到文档中，为内部使用
        if( !type || type === "fx" )
            fn = queue[0];
        if( fn !== undefined )
            fn.call(elem);
    }
})
```

fx模块animate方法的调用示例如下。

```
//each是并行处理多个动画,queue是一个接一个处理多个动画
this[ optall.queue === false ? "each" : "queue" ](function(){ /
*略*/})
```

在元素上添加自定义属性，还会引发一个问题。如果我们对这个元素进行拷贝，就会将此属性也复制过去，导致两个元素都有相同的UUID值，出现数据被错误操作的情况。jQuery早期的复制节点实现非常简单，如果元素的cloneNode方法不会复制事件，就使用cloneNode，否则使用元素的outerHTML或父节点的innerHTML，用clean方法解析一个新元素出来。但outerHTML与innerHTML都会将属性写在里面，因此需要用正则把它们清除掉。

```
//jQuery1.3.2 core.js clone方法
var ret = this.map(function(){
    if ( !jQuery.support.noCloneEvent && !jQuery.isXMLDoc(this)
 ) {
        var html = this.outerHTML;
        if ( !html ) {
            var div = this.ownerDocument.createElement("div");
            div.appendChild( this.cloneNode(true) );
            html = div.innerHTML;
        }

        return jQuery.clean([html.replace(/ jQuery\d+="(?:\d+|n
ull)"/g, "").replace (/^\s*/, "")])[0];
    } else
        return this.cloneNode(true);
});
```

在jQuery 1.4中，我们发现它对**object** 、**embed** 、**applet** 这3种元素进行了特殊处理，缘由这3个元素是用于加载外部资源的，比如flash、silverlight、media play、realone player、window自带的日历组件、颜色选择器等。在旧版本IE中，元素节点只是COM的包装，一旦引入这些资源后，它就会变成那种资源的实例。一旦这资源是由VB等语言写的，由于VM有严格的访问控制，不能随便给对象添加新属性与方法，就会遇到抛错的可能。因此jQuery做出一个决定，对于这3种元素，就不为它们缓存数据。jQuery在内部弄了一个叫noData的对象，专门放置它们的tagName。

```
noData: {
    "embed": true,
    "object": true,
    "applet": true
},
//代码防御
if ( elem.nodeName && jQuery.noData[elem.nodeName.toLowerCase()
] ) {
    return;
}
```

jQuery1.4还对$.data进行了改进，允许第二个参数为对象，方便储存多个数据。UUID对应的自定义属性expando也放进命名空间之下了。queue与dequeue方法被剥离成一个新模块。

jQuery1.43带来3项改进。

首先是添加changeData自定义方法，不过这套方法没有什么销量，只是产品经理的自恋吧。

其次，检测元素节点是否支持添加自定义属性的逻辑被独立成一个叫acceptData的方法。因为jQuery团队发现当object标签加载flash资源时，它还是可以添加自定义属性的，于是决定对这种情况网开一面。IE在加载flash时，需要对object指定一个叫classId的属性，值为**clsid:D27CDB6E- AE6D-11cf-96B8-444553540000**，因此检测逻辑就变得非常复杂，由于data、removeData都要用到，独立出来可有效节省比特。

最后HTML5对人们随便添加自定义属性的行为做出回应，新增一种叫data-* 的缓存机制。当用户设置的属性以data- 开头，它们会被保存到元素节点的dataset对象上。于是jQuery团队又有一个主意，允许人们通过设置data-* 来配置UI组件，于是他们对data-* 进行如下增强：当用户第一次访问此元素节点，会遍历它所有data- 开头的自定义属性（为了照顾旧版本IE，不能直接遍历dataset），把它们放到jQuery的缓存体中。那么当用户取数据时，会先从缓存系统中获取，没有再使用setAttribute访问"data-"自定义属性。但HTML5的缓存系统非常弱，只能保存字符串（这当然是出于循环引用的考量），于是jQuery会将它们还原为各种数据类型，如“null ”“false ”“true ”变成null、false、true，符合数字格式的字符串会转换成数字，如果它是以{ 开头 以} 结尾则尝试转成一个对象。

```
//jQuery1.43 $.fn.data
rbrace = /^(?:\{.*\}|\[.*\])$/;
if ( data === undefined && this.length ) {
    data = jQuery.data( this[0], key );
    if ( data === undefined && this[0].nodeType === 1 ) {
        data = this[0].getAttribute( "data-" + key );

        if ( typeof data === "string" ) {
            try {
                data = data === "true" ? true :
                    data === "false" ? false :
                    data === "null" ? null :
                    !jQuery.isNaN( data ) ? parseFloat( data )
:
                    rbrace.test( data ) ? jQuery.parseJSON( dat
a ) :
                    data;
            } catch( e ) {}

        } else {
            data = undefined;
        }
    }
}
```

jQuery在1.42版已经打败了Prototype.js，导致用户量暴增。它的重点改为提升性能，进入FBUG阶段（用户多，相当于免费的测试员就越多，测试覆盖面就越大），所以jQuery1.5也带来了3项改进。

首先是改进了expando。原来expando是基于时

间截，现在是版本号加随机数。因此用户可能在一个页面引入多个版本的jQuery。

其次，是否有此数据的逻辑被抽出成一个hasData方法，处理HTML5的"data-*"属性也被抽出成一个私有方法dataAttr。它们都是为了逻辑显得更清晰。

最后是dataAttr使用了JSON.parse。由于这个JSON可能是JSON2.js引入的，而JSON2.js有个非常糟糕的地方，就是为一系列原生类型添加了toJSON方法，导致for in循环判定是否为空对象出错。jQuery被逼搞了个isEmptyDataObject方法做处理。

jQuery的数据缓存系统本来就是为事件系统服务而分化出来的，到后来，它是内部众多模块的基础设施。换言之，它是供框架自己内部使用的，但一旦它公开到文档中，不可避免地，用户会使用data方法来保存他们在工作业务中用到的数据。因

此，这两类数据可能就有互相覆盖的危险。私有数据（框架使用的）与用户数据（用户自己使用的），你不能设置一个优先级来阻止它们的互相覆盖，因为没有阻止私有数据，可能框架的一些部件就不能运作，比如事件系统在每个元素的缓存体上设置的events对象。而你让用户设置的数据莫名其妙不能生效，这也是无法让人接受的。因此早期jQuery作出的让步是，框架使用的私有数据的属性名会尽可能的生癖复杂，尽量减少重名的可能，比如`__class__`、`__change__`、`_submit_attached`、`_change_attached`。

但当jQuery UI越来越庞大时，它们对数据缓存的依赖也愈发严重。同样的压力也来自普通用户。当jQuery成为世界级的著名框架后，用户数据与私有数据发生冲突的机率也就大大增加。jQuery1.5对缓存体进行改造。原来就是一个对象，现在什么数据都往里面抛，它就在这个缓存体内开辟了一个子

对象，键名为随机的`jQuery.expando`值，如果是私有数据就存到里面去。但events私有数据为了向前兼容起见，还是直接放到缓存体之上。至于，如何区分私有数据，非常简单，直接在data方法添加第4个参数，真值时为私有数据。removeData时也相应提供第3个参数，用于删除私有数据。还新设了一个_data方法，专门用于操作私有数据。图 7-1 就是缓存体的结构图。

```
var cache = {
     jQuery14312343254:{/*放置私有数据*/},
     events: {/*放置事件名与它对应的回调列表*/}
     /*这里放置用户数据*/
}
```

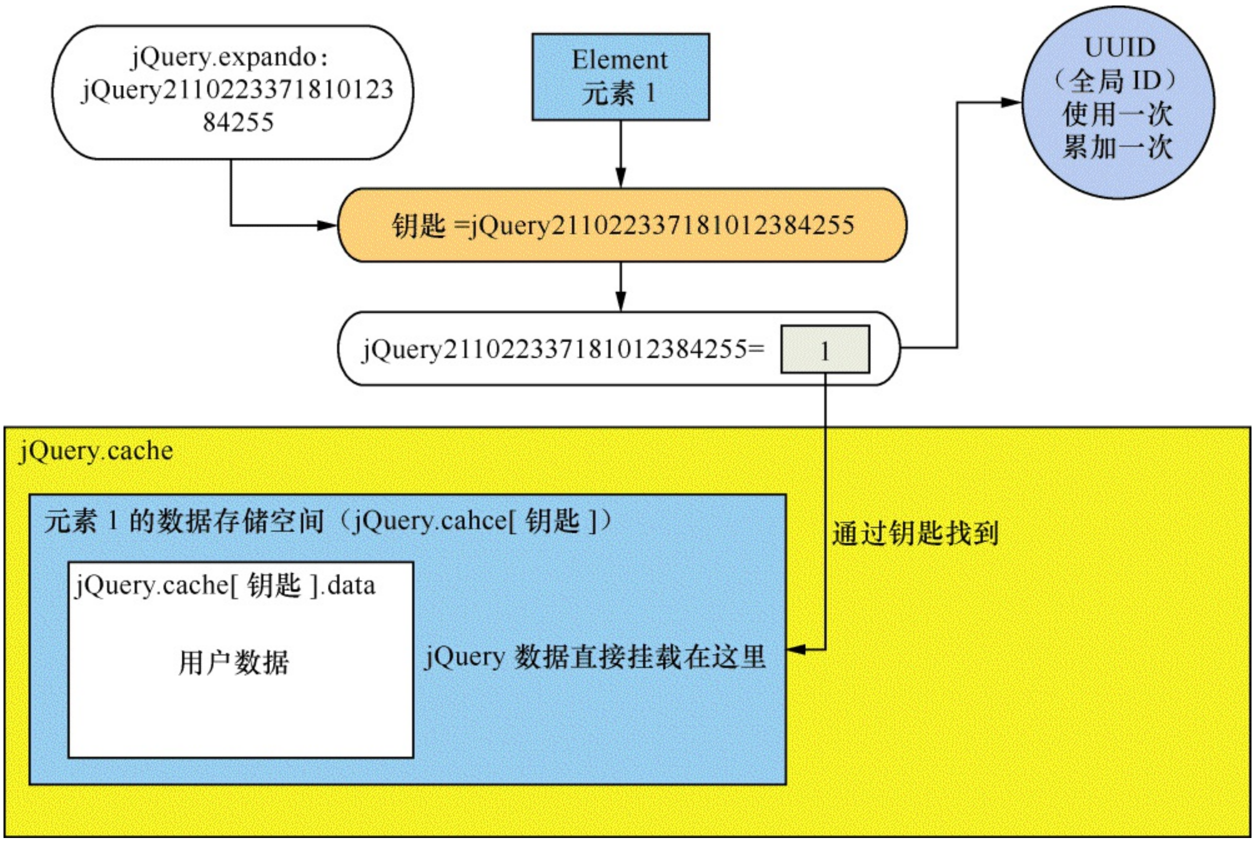


图7-1

jQuery1.7对缓存体做了改进，系统变量便放置在data对象中，为此判定缓存体为空也要做相应的改进，现在要跳过toJSON与data，新结构如下。

```
var cache = {
     data:{/*放置用户数据*/}
     /*这里放置私有数据*/
}
```

jQuery 1.8曾添加了一个叫**deleteIds** 的数组，用于重用UUID，但昙花一现。UUID的值从jQuery 1.8起就不用jQuery.uuid了，改用jQuery.guid递增生成。现在光是缓存系统就是一个庞大家族，如图7-2所示。



图7-2

7.2 jQuery的第2代缓存系统

第2代缓存系统的作者为Rick Waldron。依照他的话，要实现以下6个目标。

（1）在接口与语义上兼容1.9.x分支。

（2）通过简化储存路径为统一的方式来提高维护性。

（3）使用相同的机制来实现“私有”与“用户”数据。

（4）不再把私有数据与用户数据混在一起。

（5）不再在用户对象上添加自定义属性。

（6）方便以后可以平滑地利用WeakMap对对象进行升级（WeakMap相关规范大致在2014年完成）。

jQuery第2代缓存系统的实现方法是**valueOf** 重写！具体原理是，如果目标对象的valueOf传入一个特殊的对象，那么它就返回一个UUID，然后通过UUID在Data实例的cache对象属性上开辟缓存体。这样一来，我们就不用区分它是window对象，还是使用windowData来做替身了。另外，我们也不用顾忌embed、object、applet这3种在IE下可能无法设置私有属性的元素节点，达成第2和第5个目标。

在第1代缓存系统中，每个缓存体的结构为一个拥有data对象属性的对象，data对象属性里面放用户数据，而缓存体的其他键值用于对应私有数据。第2代则在框架内部添加了一个Data类，它的实例有一个cache属性，私有数据与用户数据分别由一个Data实例来维护，这就达成第3和第4个目标。

```
function Data() {
    this.cache = {};
}

Data.uid = 1;
```

```
Data.prototype = {
    locker: function(owner) {
        var ovalueOf,
        //owner为元素节点、文档对象、window对象
        //由于浏览器的差异性，首先我们检测一下它们valueOf方法有没有被重写
        //我们通过觉得此3类对象的构造器进行原型重写的成本过大，只能对每一个实例的valueOf方法进行重写
        //检测方式为传入Data类，如果是返回"object"说明没有被重写，返回"string"则是被重写
        //这个字符串就是我们上面所说的UUID，用于在缓存仓库上开辟缓存体
        unlock = owner.valueOf(Data);
        //这里的重写使用了 Object.defineProperty方法，因为在这个版本jQuery不打算往下兼容IE6～IE8
        //Object.defineProperty的第3个参数为对象，如果不显示设置enumerable、writable、configurable，
        //则会默认为false。这也正如我们所期待的那样，我们不再希望人们来遍历它，重写它，再次动它的配置
        //这个过程被jQuery称之为开锁，通过valueOf这扇大门，进入到仓库
        if(typeof unlock !== "string") {
            unlock = jQuery.expando + Data.uid++;
            ovalueOf = owner.valueOf;

            Object.defineProperty(owner, "valueOf", {
                value: function(pick) {
                    if(pick === Data) {
                       return unlock;
                    }
                    return ovalueOf.apply(owner);
                }

           });
        }
        //接下来就是开辟缓存体
        if(!this.cache[unlock]) {
            this.cache[unlock] = {};
        }

        return unlock;
    },
    set: function(owner, data, value) {
        //写方法
        var prop, cache, unlock;
        //得到UUID与缓存体
        unlock = this.locker(owner);
        cache = this.cache[unlock];
```

```
        //如果传入3个参数，第2个为字符串，那么直接在缓存体上添加新的键值对
        if(typeof data === "string") {
            cache[data] = value;
            //如果传入2个参数，第2个为对象
        } else {
            //如果缓存体还没有添加过任何对象，那么直接赋值，否则使用for
in 循环添加新键值对
            if(jQuery.isEmptyObject(cache)) {
                cache = data;
            } else {
                for(prop in data) {
                    cache[prop] = data[prop];
                }
            }
        }
        this.cache[unlock] = cache;

        return this;
    },
    get: function(owner, key) {
        //读方法
        var cache = this.cache[this.locker(owner)];
        //如果只有一个参数，则返回整个缓存体
        return key === undefined ? cache : cache[key];
    },
    access: function(owner, key, value) {
        //决定是读方法或是写方法，然后做相应操作
           if (key === undefined ||
                ((key && typeof key === "string") && value ===
undefined)) {
           return this.get(owner, key);
        }
        this.set(owner, key, value);
        return value !== undefined ? value : key;
    },
    remove: function(owner, key) {
        //略，与第1代差不多
    },
    hasData: function(owner) { //判定此对象是否缓存了数据
        return !jQuery.isEmptyObject(this.cache[this.locker(own
er)]);
    },
    discard: function(owner) { //删除它的用户数据与私有数据
        delete this.cache[this.locker(owner)];
    }
```

```
};
var data_user, data_priv;

function data_discard(owner) {
    data_user.discard(owner);
    data_priv.discard(owner);
}

data_user = new Data();
data_priv = new Data();
```

接下来就简单了，暴露给用户调用的方法都只是一个空壳，用来转交给data_user、date_priv这两个实例对象处理，并且私有数据的处理再也不通过用户数据的渠道了。

```
jQuery.extend({
    // UUID
    expando: "jQuery" + ( core_version + Math.random() ).replac
e( /\D/g, "" ),

    //用于向前兼容
    acceptData: function() {
        return true;
    },

    hasData: function( elem ) {//判定是否缓存了数据
        return data_user.hasData( elem ) || data_priv.hasData(
elem );
    },

    data: function( elem, name, data ) {//读写用户数据
        return data_user.access( elem, name, data );
    },
```

```
    removeData: function( elem, name ) {//删除用户数据
        return data_user.remove( elem, name );
    },
    _data: function( elem, name, data ) {//读写私有数据
        return data_priv.access( elem, name, data );
    },

    _removeData: function( elem, name ) {//删除私有数据
        return data_priv.remove( elem, name );
    }
});
```

重写valueOf这一招的确漂亮，因此任何非纯空对象都有valueOf方法，然后通过闭包保存用于关联缓存仓库的UUID。但闭包也意味着特吃内存，这是此系统的最大缺憾。

7.3 jQuery的第3代缓存系统

jQuery第3代是基于Object.defineProperty，并且为了兼容jQuery 1.9，只改动**Data**对象，外部API一律不动，如表7-1所示。

表**7-1**

	开创缓存空间	添加数据	获取数据	删除数据	读写数据	判定存在性
2	locker	set	get	remove	access	hasData
3	cache	set	get	remove	access	hasData

```
function Data() {
    this.expando = jQuery.expando + Data.uid++;
}
Data.uid = 1;
Data.prototype = {
    cache: function( owner ) {
        // 从目标对象（可能是元素节点、window、document）
        //访问一个特殊的属性，判定其否为一个对象
        var value = owner[ this.expando ];
        if ( !value ) {
            value = {}
            // 目标只能是上面提到的3种对象才进入此分支
            if ( acceptData( owner ) ) {

                //如果是节点类型（为了兼并IE）
```

```
                if ( owner.nodeType ) {
                    owner[ this.expando ] = value;
                } else {
                    //在window上创建一个不可遍历的对象属性
                    Object.defineProperty( owner, this.expando,
 {
                       value: value,
                       configurable: true
                    } )
                }
           }
        }
        return value;
    },
    set: function( owner, data, value ) {
        var prop,
            cache = this.cache( owner );

        //如果是3个参数的情况
        if ( typeof data === "string" ) {
            cache[ jQuery.camelCase( data ) ] = value;
        } else {
            // 如果是2个参数
            for ( prop in data ) {
                 cache[ jQuery.camelCase( prop ) ] = data[ prop
 ];
            }
        }
        return cache;
    },
    get: function( owner, key ) {
        //与第2代相仿
    },
    access: function( owner, key, value ) {
        //与第2代相仿
    },
    remove: function( owner, key ) {
        //与第2代相仿
    },
    hasData: function( owner ) {
        var cache = owner[ this.expando ];
        return cache !== undefined && !jQuery.isEmptyObject( ca
che );
    }
};
```

jQuery为了照顾老旧版本，之前承诺的WeakMap方案一直没有上架，不能不说是遗憾。如果直接上WeakMap，jQuery的数据缓存系统可就精简许多。

7.4　有容量限制的缓存系统

上面提到的jQuery数据缓存系统是纯粹为DOM服务的，当我们的框架出现分层架构，就需要另一种缓存系统了。因为页面上的节点数量总在可控范围，所以不会有将缓存系统撑死的情况。但如果要保持的是普通JavaScript对象，就不一定了。如果这个是前后通吃的框架，并且它的缓存系统要为上百万用户服务时，就更需要加限制。

一个简单的带容量限制的缓存系统可以短成这样。

```
//https://github.com/jquery/sizzle/blob/master/src/sizzle.js
function createCache(size) {
    var keys = []
    function cache( key, value ) {
        // 避开IE6～IE8的toString和valueOf方法，
        // 及cache本身的name、length属性
        if ( keys.push( key + " " ) > size) {
            delete cache[ keys.shift() ];
        }
        return (cache[ key + " " ] = value);
    }
    return cache;
}
```

```
var cache = createCache(100)
cache("aaa", "bbb")
```

此缓存体完全相信是jQuery的哈希寻找算法实现的，当一个缓存体的键值对数量非常庞大，要命中一个键名就可能很花时间。于是我们需要求助Least Recently User、Least Recently User2、Two Queues、Adaptive Replacement Cache等缓存算法。

其实从实用性与行数来考虑，上面这些算法，就数LRU最受青睐。下面给出avalon的现役缓存系统，其核心是一个链表与hashmap。

```
// https://github.com/rsms/js-lru
function LRU(maxLength) {
    this.size = 0
    this.limit = maxLength
    this.head = this.tail = void 0
    this._keymap = {}
}

var p = LRU.prototype

p.put = function (key, value) {
    var entry = {//创建一个结点
        key: key,
        value: value
    }
```

```
    this._keymap[key] = entry
    if (this.tail) {//调整链表
        this.tail.newer = entry
        entry.older = this.tail
    } else {
        this.head = entry
    }
    this.tail = entry
    if (this.size === this.limit) {
        this.shift()
    } else {
        this.size++
    }
    return value
}

p.shift = function () {
    var entry = this.head
    if (entry) {
        this.head = this.head.newer
        this.head.older =
                entry.newer =
                entry.older =
                this._keymap[entry.key] = void 0
        delete this._keymap[entry.key]
    }
}
p.get = function (key) {
    var entry = this._keymap[key]
    if (entry === void 0)
        return
    if (entry === this.tail) {
        return  entry.value
    }
    // HEAD--------------TAIL
    //   <.older   .newer>
    //  <--- add direction --
    //   A  B  C  <D>  E
    if (entry.newer) {
        if (entry === this.head) {
            this.head = entry.newer
        }
        entry.newer.older = entry.older // C <-- E.
    }
    if (entry.older) {
```

```
        entry.older.newer = entry.newer // C. --> E
    }
    entry.newer = void 0 // D --x
    entry.older = this.tail // D. --> E
    if (this.tail) {
        this.tail.newer = entry // E. <-- D
    }
    this.tail = entry
    return entry.value
}

module.exports = LRU
//=======

var a = new Cache(100)
a.put("aaa","bbb")
console.log(a.get("aaa"))
```

7.5　本地存储系统

上面提到的缓存系统都是基于内存的，当页面跳转时就什么都没有了。因此我们需要将数据序列化到本地，方便下次打开时再使用。传统方式是使用cookie，但cookie问题很多。

（1）cookie数量和长度的限制。每个domain最多只能有20条cookie，每个cookie长度有4096字节的限制。尽管在当今新的浏览器和客户端设备版本中，支持8192字节的Cookie大小已愈发常见。

（2）用户能配置为禁用cookie。

（3）cookie每次随HTTP事务一起发送，占用带宽。

（4）潜在的安全风险。cookie可能会被篡改，用户可能会操纵其计算机上的cookie，这意味着会

对安全性造成潜在风险或者导致依赖于cookie的应用程序失败。另外，虽然cookie只能被发送到客户端的域访问，但是历史上的黑客已经发现从用户计算机上的其他域访问cookie的方法。您可以手动加密或解密cookie，但这需要额外的编码，并且加密或解密需要耗费一定的时间进而影响应用程序的性能。

为弥补cookie缺陷，浏览器厂商又推出了其他方案，如图7-3所示。

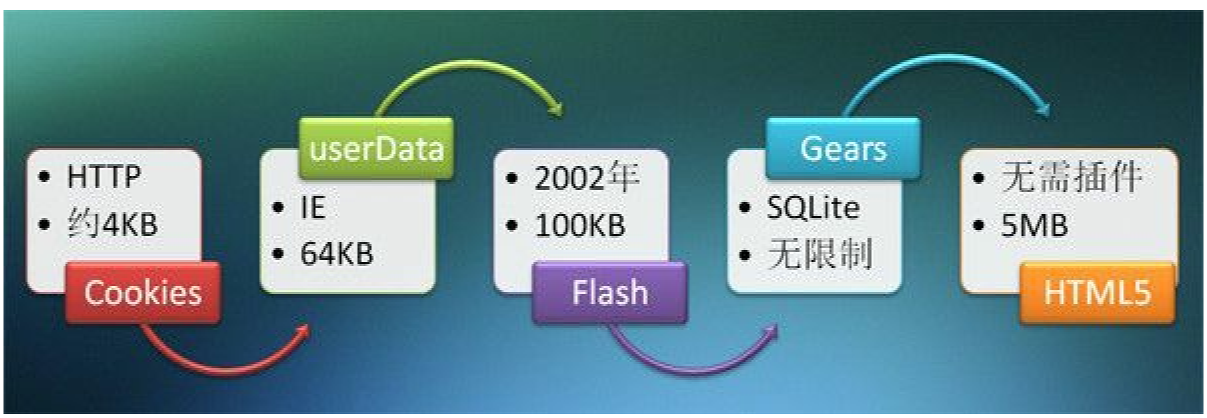


图7-3

userData是IE的东西，“垃圾”。现在用的最多

的是Flash吧，空间是Cookie的25倍，基本够用。再之后Google推出了Gears，虽然没有限制，但不爽的地方就是要装额外的插件（没具体研究过）。到了HTML5把这些都统一了，官方建议是每个网站5MB，非常大了，就存些字符串，足够了。比较诡异的是，居然所有支持的浏览器目前都采用的5MB，尽管有一些浏览器可以让用户设置，但对于网页制作者来说，目前的形势就5MB来考虑是比较妥当的，如图7-4所示。

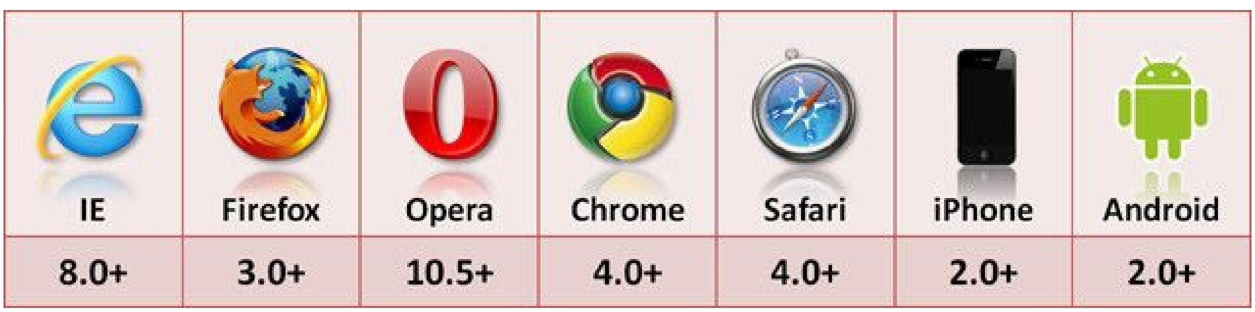


图7-4

IE在8.0的时候就支持了，非常出人意料。不过需要注意的是，IE、Firefox测试的时候需要把文件上传到服务器上（或者localhost），直接点开本地

的HTML文件，是不行的。

localStorage的缺点。

（1）localStorage大小限制在500万字符左右，各个浏览器不一致。

（2）localStorage在隐私模式下不可读取。

（3）localStorage本质是在读写文件，数据多的话会比较慢（Firefox会一次性将数据导入内存，想想就觉得吓人）。

（4）localStorage不能被爬虫爬取，不要用它完全取代URL传参，瑕不掩瑜。

以上问题皆可避免，所以我们的关注点应该放在如何使用localStorage。

localStorage的API：

```
function getLocalStorage() {
    //现在的一些浏览器有一种功能叫无痕浏览，
    //此模式localStorage会被禁用了,但对象还在,此调用其方法时会抛异常
    if (window.localStorage) {
        try {
            localStorage.setItem("key", "value");
            localStorage.removeItem("key");
            return localStorage
        } catch (e) {
        }
    }
}
var db = getLocalStorage()
if (db) {
    db.setItem('author', 'jasonling')
    db.setItem('company', 'Tencent')
    db.setItem('introduction', 'A code lover !')
    alert(localStorage['author'])
    alert(db.getItem('company'))
    //删除
    db.removeItem('company')
    alert(db.getItem('company'))
    //当然也可以用db.setItem('company', '');来删除一个值，但这样删不彻
底。
    //清除
    db.clear()
    alert(localStorage['author']) //undefined
}
```

下面我们讲解一下store.js这个库。它是一个兼容所有浏览器的localStorage适配器，不需要借助cookie或者Flash。store.js会根据浏览器自动选择使用localStorage或者userData来实现本地存储功能如表7-2所示。

表**7-2**

get	getAll	set	has	remove	clear	forEach
获取	获取所有	添加	存在性	移除	清空	遍历

```
"use strict"

module.exports = (function() {
    // Store.js
    var store = {},
        win = (typeof window != 'undefined' ? window : global),
        doc = win.document,
        localStorageName = 'localStorage',
        scriptTag = 'script',
        storage

    store.disabled = false
    store.version = '1.3.20'
    //定义接口（空实现，即便浏览器不支持userData与localStorage也不会报
错）
    store.set = function(key, value) {}
    store.get = function(key, defaultVal) {}
    store.has = function(key) { return store.get(key) !== undef
ined }
    store.remove = function(key) {}
    store.clear = function() {}
    store.transact = function(key, defaultVal, transactionFn) {
        if (transactionFn == null) {
            transactionFn = defaultVal
            defaultVal = null
        }
        if (defaultVal == null) {
            defaultVal = {}
        }
        var val = store.get(key, defaultVal)
        transactionFn(val)
        store.set(key, val)
    }
    store.getAll = function() {
```

```
        var ret = {}
        store.forEach(function(key, val) {
            ret[key] = val
        })
        return ret
    }
    store.forEach = function() {}
    //内部方法
    store.serialize = function(value) {
        return JSON.stringify(value)
    }
    //内部方法
    store.deserialize = function(value) {
        if (typeof value != 'string') { return undefined }
        try { return JSON.parse(value) }
        catch(e) { return value || undefined }
    }

    // 判定localStorage是否可用，包括禁用的情况
    function isLocalStorageNameSupported() {
        try { return (localStorageName in win && win[localStora
g
eName]) }
        catch(err) { return false }
    }
    //使用localStorage实现本库
    if (isLocalStorageNameSupported()) {
        storage = win[localStorageName]
        //添加（如果存在同名数据会覆盖）
        store.set = function(key, val) {
            if (val === undefined) { return store.remove(key) }
            storage.setItem(key, store.serialize(val))
            return val
        }
        //获取，可以设置备用值
        store.get = function(key, defaultVal) {
            var val = store.deserialize(storage.getItem(key))
            return (val === undefined ? defaultVal : val)
        }
        //移除
        store.remove = function(key) { storage.removeItem(key)
}
        //清空
        store.clear = function() { storage.clear() }
        //遍历
```

```
        store.forEach = function(callback) {
            for (var i=0; i<storage.length; i++) {
                 var key = storage.key(i)
                 callback(key, store.get(key))
            }
        }
    //如果浏览器支持addBehavior，说明也支持userData
    } else if (doc && doc.documentElement.addBehavior) {
        var storageOwner,
            storageContainer
        //userData的宿主对象为一个元素节点，
        //使用userData后，IE会重写该节点的getAttribute方法，只支持一个
参数
        //而getAttribute在IE6～IE8要获取A标签的完整路径，需要用到其第2
个参数
        //http://www.cnblogs.com/_franky/archive/2010/10/25/186
0649.html
        //为了不影响现有节点，我们可以用ActiveXObject(htmlfile)
        //创建一个看不见的HTML文档，在里面创建一个div。

        //userData还要指定路径，这时我们使用 /favicon.ico 这个神奇的地
址，即便返回404也不会产生危害
        // (see: http://msdn.microsoft.com/en-us/library/aa7525
74(v=VS.85).aspx)
        try {
            storageContainer = new ActiveXObject('htmlfile')
            storageContainer.open()
            storageContainer.write('<'+scriptTag+'>document.w=w
indow</'+scriptTag+'>
            <iframe src="/favicon.ico"></iframe>')
            storageContainer.close()
            storageOwner = storageContainer.w.frames[0].documen
t
            storage = storageOwner.createElement('div')
        } catch(e) {

            storage = doc.createElement('div')
            storageOwner = doc.body
        }
        var withIEStorage = function(storeFunction) {
            return function() {
                 var args = Array.prototype.slice.call(argument
s, 0)
                 args.unshift(storage)
                 // See <a>http://msdn.microsoft.com/en-us/libr
```

```
ary/m</a>
s531081(v=VS.85).aspx
                 // and <a>http://msdn.microsoft.com/en-us/libr
ary/m</a>
s531424(v=VS.85).aspx
                 storageOwner.appendChild(storage)
                 storage.addBehavior('#default#userData')
                 storage.load(localStorageName)
                 var result = storeFunction.apply(store, args)
                 storageOwner.removeChild(storage)
                 return result
            }
        }

        // IE7对键名有特殊要求，我们需要转译一下
        // See https://github.com/marcuswestin/store.js/issues/
40
        // See https://github.com/marcuswestin/store.js/issues/
83
        var forbiddenCharsRegex = new RegExp("[!\"#$%&'()*+,/\\
\\:;<=>?@[\\]^`{|}~]", "g")
        var ieKeyFix = function(key) {
            return key.replace(/^d/, '___$&').replace(forbidden
CharsRegex, '___')
        }
        //设置
        store.set = withIEStorage(function(storage, key, val) {
            key = ieKeyFix(key)
            if (val === undefined) { return store.remove(key) }
            storage.setAttribute(key, store.serialize(val))
            storage.save(localStorageName)
            return val
        })
        //添加
        store.get = withIEStorage(function(storage, key, defaul
tVal) {
            key = ieKeyFix(key)
            var val = store.deserialize(storage.getAttribute(ke
y))
            return (val === undefined ? defaultVal : val)
        })
        //移除
        store.remove = withIEStorage(function(storage, key) {
            key = ieKeyFix(key)
            storage.removeAttribute(key)
```

```
            storage.save(localStorageName)
        })
        //清空
        store.clear = withIEStorage(function(storage) {
            var attributes = storage.XMLDocument.documentElemen
t.attributes
            storage.load(localStorageName)
            for (var i=attributes.length-1; i>=0; i--) {
                storage.removeAttribute(attributes[i].name)
            }
            storage.save(localStorageName)
        })
        store.forEach = withIEStorage(function(storage, callbac
k) {
            var attributes = storage.XMLDocument.documentElemen
t.attributes
            for (var i=0, attr; attr=attributes[i]; ++i) {
                callback(attr.name, store.deserialize(storage.g
e
                tAttribute(attr.name)))
            }
        })
    }

    try {
        var testKey = '__storejs__'
        store.set(testKey, testKey)
        if (store.get(testKey) != testKey) { store.disabled = t
rue }
        store.remove(testKey)
    } catch(e) {
        store.disabled = true
    }
    store.enabled = !store.disabled

    return store
}())
```

隐私模式下可以采用window.name模拟

localStorage的方式处理，因为window.name在载入新页面或刷新后，其值依然是上次页面设置的值。相关资料：

```
https://github.com/kbjr/Box.js
https://www.baidufe.com/item/af0bb5872f2a1ef337ce.html
```

7.6　总结

本章节讲解了两种存储系统，基于内存的与基于本地储存的，它们间可以相互转换相互支撑，是我们构建Web APP的基石。

第8章　样式模块



样式模块大致分为两大块，精确获取样式值与设置样式、精确是用于修饰获取的。由于样式分为

外部样式、内部样式与行内样式，再加个important对选择器权重的干扰，我们实际很难看到元素是应用了哪些样式规则。因此样式模块，80%的比重在于获取这一块，像 offset、滚动条也归入这一块。

大体上，我们在标准浏览器下是使用getComputedStyle，IE6～IE8下使用currentStyle来获取元素的精确样式。不过getComputedStyle并不挂在元素上，而是window的一个API，它返回一个对象，可以选择使用getPropertyValue方法传入连字符风格的样式名取得其值，或者属性法+驼峰风格的样式名去取值，但考虑到currentStyle也是使用属性法+驼峰风格，我们就统一使用后者。

```
var getStyle = function(el, name) {
    if (el.style) {
        name = name.replace(/\-(\w)/g, function(all, letter) {
            return letter.toUpperCase();
        });
        if (window.getComputedStyle) {
            //getComputedStyle的第二个伪类是用于对付伪类的，如滚动条，
placeholder，
            //但IE9不支持，因此我们只管元素节点，上面的el.style过滤掉了
            return el.ownerDocument.getComputedStyle(el, null)[
 name ]
```

```
        } else {
            return el.currentStyle[ name ];
        }
    }
}
```

设置样式则更没难度，直接el.style[name] = value搞定。

但框架要考虑的东西很多，如兼容性、易用性、扩展性。

（1）样式名要同时支持连字符风格（CSS的标准风格）与驼峰风格（DOM的标准风格）。

（2）样式名要进行必要的处理，如float样式与CSS3带私有前缀的样式（易用性的考虑）。

（3）设置样式时，对于长度宽度可以考虑直接处理数值，由框架智能补上"px"单位（易用性的考虑）。

（4）对个别样式的特殊处理，如IE下的z-index、opacity、user-select、background-position、top、left（扩展性的考虑，这里要引入插件机制或适配器机制）。

本章将围绕avalon的css模块展开，涵盖的内容大致相当于jQuery的css、offset、demensions模块，或相当于EXT4的Element.style、Element.scroll、Element.position模块。

8.1 主体架构

avalon处理样式相关的方法如图8-1所示。

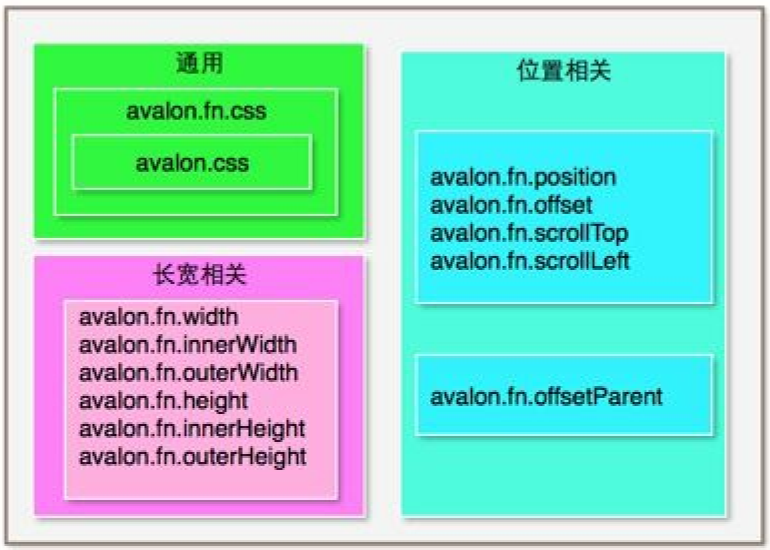


图8-1

那些位于avalon.fn中的为原型方法，暴露给一般用户使用。其中avalon.fn.css是最重要的方法，但avalon.fn.css只是一个空壳，它实现的是avalon.css的封装，只是对传参做初步分解。

```
avalon.fn.css = function (name, value) {
    if (avalon.isPlainObject(name)) {
        //如果是传入一个对象，那么进入批量set操作
```

```
        for (var i in name) {
            avalon.css(this, i, name[i])
        }
    } else {
        var ret = avalon.css(this, name, value)
    }
    return ret !== void 0 ? ret : this
}
```

avalon.css会对参数进一步分解，首先对属性名（通过avalon.cssName）进行转换，然后根据参数的类型与个数，进入3个分支处理，如图8-2所示。

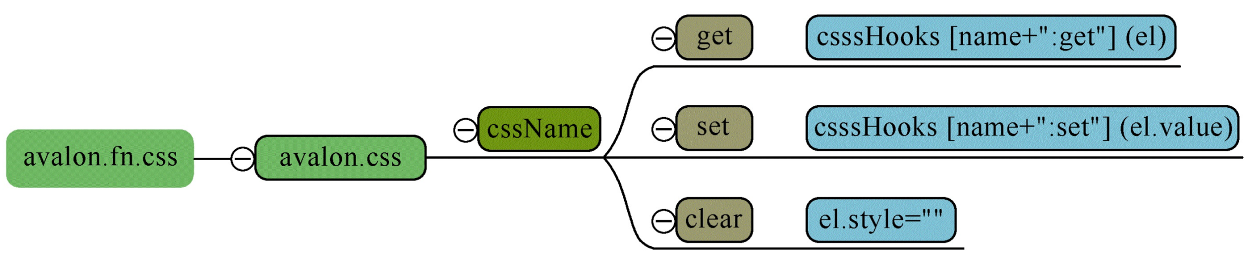


图8-2

```
avalon.css = function (node, name, value) {
     //读写删除元素节点的样式
     if (node instanceof avalon) {
         node = node[0]// 取得元素节点
     }
     var prop = avalon.camelize(name), fn
     name = avalon.cssName(prop) || prop
     //获取样式(value为布尔表示去掉单位,返回纯数值)
     if (value === void 0 || typeof value === 'boolean') {
         fn = cssHooks[prop + ':get'] || cssHooks['@:get']
         if (name === 'background') {
             name = 'backgroundColor'
         }
```

```
        var val = fn(node, name)
        return value === true ? parseFloat(val) || 0 : val
    } else if (value === '') { //清除样式
        node.style[name] = ''
    } else { //设置样式
        if (value == null || value !== value) {
            return
        }
        if (isFinite(value) && !avalon.cssNumber[prop]) {
            value += 'px'//添加单位
        }
        fn = cssHooks[prop + ':set'] || cssHooks['@:set']
        fn(node, name, value)
    }
}
```

里面用到avalon.cssName及avalon.cssHooks，一个是样式名转换函数，另一个是适配器对象，用于添加各种样式的设置或获取钩子方法。目前没有一个浏览器（包括Chrome）在没有钩子方法的情况下，能顺列操作所有样式。

avalon.cssHooks有两个默认通用方法，即cssHooks['@:get']、cssHooks['@:set']。为实现它们，在IE下都要大费周章，因为在旧式IE中，有3个地方可以存放样式值，需要交替利用它才能得到

样式的精确值，而现代浏览器下，只要一个getComputedStyle方法就行了。

```
cssHooks['@:set'] = function (node, name, value) {
    try {
        //node.style.width = NaN;node.style.width = 'xxxxxxx';
        //node.style.width = undefine 在旧式IE下会抛异常
        node.style[name] = value
    } catch (e) {
    }
}
if (window.getComputedStyle) {
    cssHooks['@:get'] = function (node, name) {
        if (!node || !node.style) {
            throw new Error('getComputedStyle要求传入一个节点 ' +
node)
        }
        var ret, styles = getComputedStyle(node, null)
        if (styles) {
            ret = name === 'filter' ? styles.getPropertyValue(n
ame) : styles[name]
            if (ret === '') {
                ret = node.style[name] //其他浏览器需要我们手动取内
联样式
            }
        }
        return ret
    }
} else {
    var rnumnonpx = /^-?(?:\d*\.)?\d+(?!px)[^\d\s]+$/i
    var rposition = /^(top|right|bottom|left)$/
    var ralpha = /alpha\([^)]*\)/i
    var ie8 = !!window.XDomainRequest
    var salpha = 'DXImageTransform.Microsoft.Alpha'
    var border = {
        thin: ie8 ? '1px' : '2px',
        medium: ie8 ? '3px' : '4px',
        thick: ie8 ? '5px' : '6px'
    }
    cssHooks['@:get'] = function (node, name) {
        //取得精确值，不过它有可能是带em、pc、mm、pt、%等单位
```

```
        var currentStyle = node.currentStyle
        var ret = currentStyle[name]
        if ((rnumnonpx.test(ret) && !rposition.test(ret))) {
            //①，保存原有的style.left, runtimeStyle.left,
            var style = node.style,
                    left = style.left,
                    rsLeft = node.runtimeStyle.left
            //②由于③处的style.left = xxx会影响到currentStyle.left
，

            //因此把它currentStyle.left放到runtimeStyle.left，
            //runtimeStyle.left拥有最高优先级，不会style.left影响
            node.runtimeStyle.left = currentStyle.left
            //③将精确值赋给到style.left，然后通过IE的另一个私有属性 s
tyle.pixelLeft
            //得到单位为px的结果；fontSize的分支见http://bugs.jquery
.com/ticket/760
            style.left = name === 'fontSize' ? '1em' : (ret ||
0)
            ret = style.pixelLeft + 'px'
            //④还原 style.left，runtimeStyle.left
            style.left = left
            node.runtimeStyle.left = rsLeft
        }
        if (ret === 'medium') {
            name = name.replace('Width', 'Style')
            //border width 默认值为medium，即使其为0'
            if (currentStyle[name] === 'none') {
                ret = '0px'
            }
        }
        return ret === '' ? 'auto' : border[ret] || ret
    }
}
```

8.2　样式名的修正

不是所有样式名都是直接用正则简单处理一下就行，这里存在3个陷阱，float对应的JavaScript属性存在兼容性问题、CSS3大爆炸时带来的私有前缀以及IE的私有前缀不合流问题。

float是一个关键字，因此不能直接用，IE这边给的替换品是styleFloat，W3C是cssFloat。

CSS3给Web开发带来了革命性的影响，以前很多需要JavaScript才能实现的复杂效果，现在使用CSS3就能简单实现。但，标准制定总是滞后于浏览器厂商的实现，只要有一个浏览器实现一个很酷的效果，其他浏览器也跟风。浏览器厂商也有先见之明，不确定自己的实现与W3C最后定案的效果是否一致，于是私有前缀便产生了。不过私有前缀是由来以久的东西，并不是CSS3在这概念被炒热时才出

来的，比如-ms-是IE8时代就存在、-khtml-就更早了。现在私有前缀在各浏览器定义如表8-1所示。

表**8-1**

浏览器	**IE**	**Firefox**	**Chrome**	**Safari**	**Opera**	**Konqueror**
前缀	-ms-	-moz-	-webkit-	-webkit-	-o-	-khtml-

2013年年初，Google嫌webkit内核太臃肿，决定自己单干，取名为blink，并在Chrome28起使用此内核。不过为了减轻用户负担，还是使用-webkit-做前缀。目前，使用-webkit-前缀的有Opera、Safari、Chrome三家。

上述的这些前缀加上样式名再驼峰化就是真正可用的样式名（对那些试验性样式来说）了，比如-ms-transform -> MsTransform、-webkit-transform -> WebkitTransform、-o-transform -> Otransform、-moz-transform -> MozTransform。但试验性样式迟

早会退出舞台的，它们会卸掉前缀重新亮相，比如FF17就直接可用transform。但光是这样是不够的，还有第3个问题，IE下-ms-transform对应的JavaScript属性名为msTransform，因此这个正则就非常复杂。我们要动用一个函数通过特性侦测手段获取它。在avalon，它叫cssName，在jQuery叫做vendorPropName。由于特性侦测是DOM操作，消耗很大，因此获取后就应缓存起来，避免重复检测，这个对象在avalon称之为cssMap。下面是对应的源码。

```
var camelize = avalon.camelize
var root = document.documentElement
var prefixes = ['', '-webkit-', '-o-', '-moz-', '-ms-']
var cssMap = {
    'float': window.Range ? 'cssFloat' : 'styleFloat'
}
avalon.cssName = function (name, host, camelCase) {
    if (cssMap[name]) {
        return cssMap[name]
    }
    host = host || root.style || {}
    for (var i = 0, n = prefixes.length; i < n; i++) {
        camelCase = camelize(prefixes[i] + name)
        if (camelCase in host) {
            return (cssMap[name] = camelCase)
        }
    }
    return null
```

```
}
```

prefixes的顺序设置得相当有技巧。” ”表示没有私有前缀，此样式已经标准化，故排在最前面。wekbit是最多浏览器使用的，因此排第2。“-o-”是最小众的，因此排末尾。其他两个夹中间。

我们通过上面的函数，只需传入一个参数，就可以得到真正可用的样式名了。

8.3　个别样式的特殊处理

现在我们来展示cssHooks的价值所在。它是专门用于对付那些有兼容性问题、不按常规出牌的奇葩样式。cssHooks为一个普通的对象，它每个属性名都以xxx+":set"或xxx+":get"命名，值为处理函数。

8.3.1　opacity

在开始之前，我们先了解一下CSS是怎么进行的吧，毕竟JavaScript设置透明度只是把这过程由死态变成动态。Firefox和webkit系浏览器的古老版本的透明度设置如下。

```
.opacity{
    -moz-opacity: 0.5;
    -khtml-opacity: .5;
}
```

有资料表明Firefox在0.9版本中声明废弃此样式，反正笔者在Firefox3.6.24与Firefox16中测试，-moz-opacity:已经没有效果了。现在标准浏览器的透明度设置（包括IE9）如下。

```
.opacity {
    opacity:.5
}
```

opacity会同时让背景与内容透明，如果想内容不透明，就要用`rgba`与`hsla`。不过它们是一种样式值的格式，并不是样式，不列入我们的讨论范围。

旧版本IE的透明度设置，依赖于私有的滤镜**DXImageTransform.Microsoft.Alpha**。

```
.opacity {
    filter: progid:DXImageTransform.Microsoft.Alpha(opacity=40)
;
}
```

不过这个太长了，IE又提供了一个简短的，也是现在主流的在IE设置透明度的方式。

```
.opacity {
     filter: alpha(opacity=40)
}
```

在IE8，开始推广-ms-私有前缀，透明滤镜的写法变成如下。

```
.opacity {
    -ms-filter: "progid:DXImageTransform.Microsoft.Alpha(opacity=40)";
/*IE8专用，必须用引号括起*/
}
```

对于IE6和IE7还需要注意一点：为了使得透明设置生效，元素必须是“有布局”。一个元素可以通过使用一些CSS属性来使其被布局，有如width和position。关于微软专有的hasLayout属性详情，以及如何触发它，大家可以网上搜索一下。

了解这些，我们在框架中实现它们就很简单了。由于opacity已经被标准浏览器所支持，我们的重心落在旧版本IE中。

先实现获取透明度，IE是使用滤镜实现透明的，不同的IE版本其滤镜的定义方式也不同，但相同点也很好找，就是透明值总是跟在`opactity=`之后。因此我们写一个正则将它们截取出来，然后转换为数字除以100。如果此元素没有使用透明滤镜，直接返回`"1"`好了。

```
//cssHooks的钩子函数方法详见第1节，依次传入元素节点，样式名，样式值
var ropactiy = /(opacity|\d(\d|\.)*)/g
cssHooks['opacity:get'] = function (node) {
     var match = node.style.filter.match(ropactiy) || []
        var ret = false
        for (var i = 0, el; el = match[i++];) {
            if (el === 'opacity') {
                ret = true
            } else if (ret) {
                return (el / 100) + ''
            }
        }
        return '1' //确保返回的是字符串
}
```

设置透明度就有点麻烦，首先用户传入的是0～1的数值，我们应用于滤镜需要放大100倍。在IE6、IE7中，我们需要判定元素有没有**hasLayout**，若没有，使用zoom =1让其hasLayout。在IE7、IE8中，如果透明度为100，会让文本模糊不清，需要清掉透明滤镜，这时我们就遇到一个问题了，一个元素上可能设置了多个滤镜，不能简单用`el.style.filter = ""`清掉，这要用正则把单个滤镜分割出来，把当中的透明滤镜去掉。如果滤镜在0～99，我们设置透明度有个窍门，如果已经存在透明滤镜，直接找到其滤镜对象，改其alpha值就行了，否则就需要小心翼翼拼字符串啦！

```
var ralpha = /alpha\([^)]*\)/i
cssHooks['opacity:set'] = function (node, name, value) {
        var style = node.style
        var opacity = isFinite(value) && value <= 1 ? 'alpha(op
ac
        ity=' + value * 100 + ')' : ''
        var filter = style.filter || '';
        style.zoom = 1
        //不能使用以下方式设置透明度
        //node.filters.alpha.opacity = value * 100
        style.filter = (ralpha.test(filter) ?
               filter.replace(ralpha, opacity) :
               filter + ' ' + opacity).trim()
```

```
        if (!style.filter) {
            style.removeAttribute('filter')
        }
    }
```

8.3.2　user-select

CSS3有一个叫user-select的样式，用于控制文本内容的可选择性。比如，拖动时，会出现文字被选中的状况，分散用户的注意力，这里就可以尝试使用此属性。在标准浏览器下，由于$.cssName的存在，一下子就找到可使用的样式名。而在旧版本IE中，没有这样的样式，是使用unselectable属性代替。不过由于unselectable不具继承性，加之子元素是位于父元素的上面，因此光是设置当前元素是不行的，要把它及其所有子孙都设置，具体代码如下。

```
cssHooks[ "userSelect:set" ] = function(node, name, value) {
    var allow = /none/.test(value) ? "on" : "",
            e, i = 0, els = node.getElementsByTagName('*');
    node.setAttribute('unselectable', allow);
    while ((e = els[ i++ ])) {
        switch (e.tagName.toLowerCase()) {
```

```
            case 'iframe' :
            case 'textarea' :
            case 'input' :
            case 'select' :
                break;
            default :
                e.setAttribute('unselectable', allow);
        }
    }
};
```

8.3.3　background-position

在旧版本IE中，IE只支持backgroundPositionX与backgroundPositionY，不支持background Position，而在FF早期的版本中（3.0之前），FF只支持backgroundPosition，不支持backgroundPositionX与backgroundPositionY。它们俩总是对着干的。不过，FF加入了Chrome引发的版本号竞赛，FF3已经很少人用了，我们还是照顾好IE6、IE7就行了。实现很简单，分别取backgroundPositionX与backgroundPositionY，然后把它们合在一起就是backgroundPosition。

```
cssHooks[ "backgroundPosition:get" ] = function( node, name, va
lue ) {
        var style = node.currentStyle;
        return style.backgroundPositionX +" "+style.backgroundP
ositionX
 };
```

8.3.4　z-index

z-index并不是一个难以理解的属性，但它却因错误的假设而使很多初级的开发人员陷入混乱。混乱发生的原因是z-index只能工作在被明确定义了absolute、fixed或relative这3个定位属性的元素中，它会让元素沿z轴进行排序（z轴的起点为其父节点所在的层，终点为屏幕）。如果为正数，则离用户更近，为负数则表示离用户更远。想象这样一个场合，一个相对定位的父节点，然后里面有N个绝对定位的子元素。如果没有指定z-index，它们的显示方式是按照出现的先后顺序排列（nextElementSibling会在previousElementSibling之上）；如果有就按z-index排列，如果是负数，标准

浏览器下会表现为元素排在其父节点的背后。当然，实际上元素的层叠顺序涉及更多东西，比如**stacking context**概念、IE下的select元素的bug、z-index为负时的浏览器差异等，如图8-3所示。

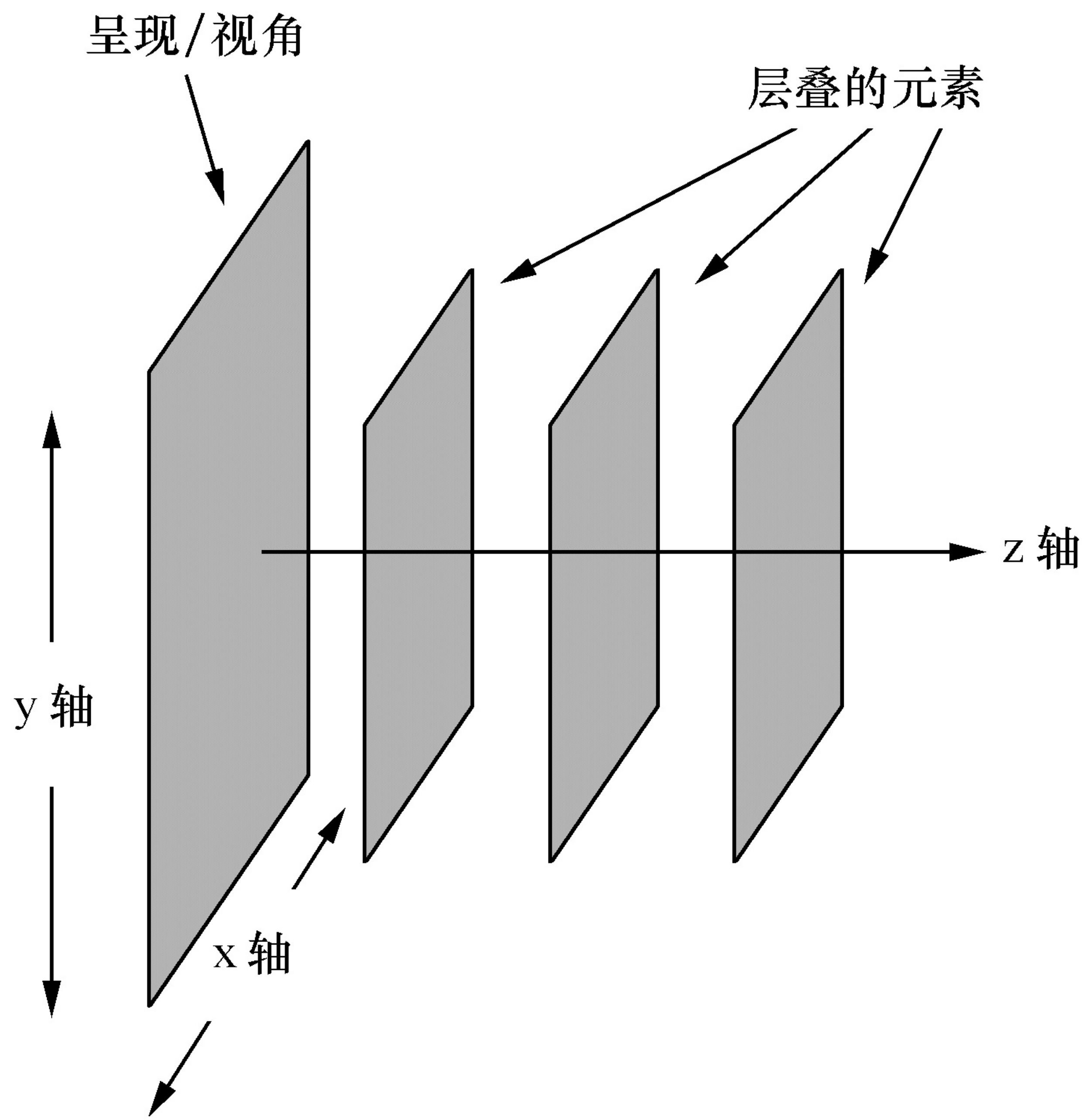


图8-3

z-index在下拉菜单、tooltip、灯箱效果、相册与拖动等中经常被使用。为了让目标控件排在最前

面，我们需要得知它们的z-index，然后有目的地改z-index或重排元素（将目标元素移出DOM树再插入父元素内部的最后一个元素之后）。

想获取z-index，这里得应对一个特殊情况，目标元素没有被定位，需要往上回溯其祖先定位元素。如果找到，就返回定位祖先的z-index值。如果最后都没找到，就返回0。

```
cssHooks["zIndex:get"] = function(node) {
    while (node.nodeType !== 9) {
    //即使元素定位了，但如果zindex设置为"aaa"这样的无效值，浏览器都会返回
auto
   //如果没有指定zindex值，IE会返回数字0，其他返回auto
        var position = getter(node, "position") || "static";
        if (position !== "static") {
            // <div style="z-index: -10;"><div style="z-index:
0;"></div></div>
            var value = parseInt(getter(node, "zIndex"), 10);
            if (!isNaN(value) && value !== 0) {
                return value;
            }
        }
        node = node.parentNode;
    }
    return 0;
};
```

8.3.5　盒子模型

在开始讲元素的尺寸位置时，我们先了解一下CSS盒子模型，如果没有这知识储备，后面就进行不下去。我们可以把页面上每一个元素节点看成一个装了东西的盒子，盒子里面的内容到盒子边框之间的距离即填充（padding），盒子本身有边框（border），用来放置子元素或文本的区域叫content，盒子边框与其他盒子之间还存在边界（margin），为了把盒子装饰得五彩缤纷，而不只是一个死板的矩形，它还有背景颜色与背景图片。为了方便渲染，它们都会于不同的层上，如图8-4所示。

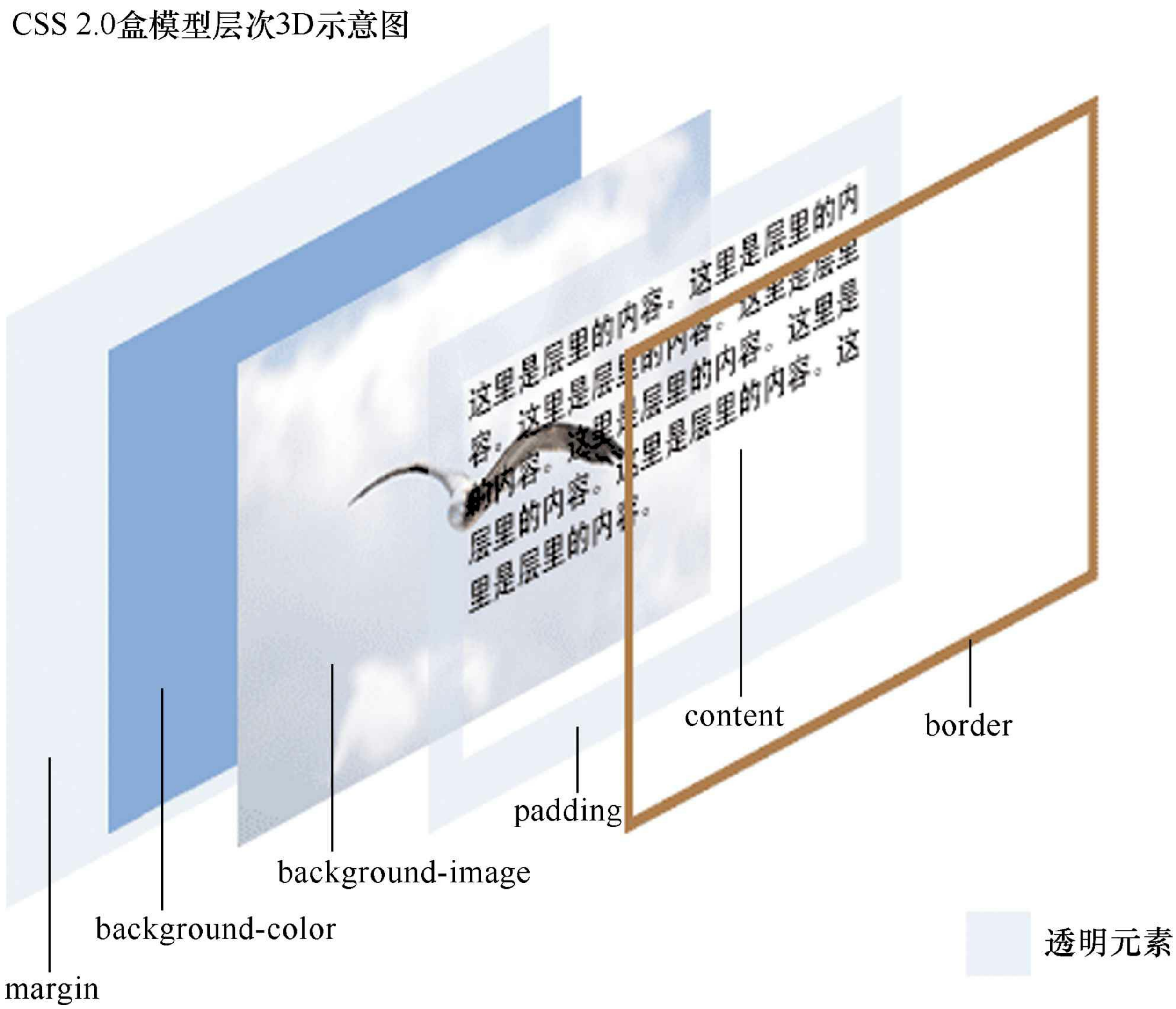


图8-4

上面显示了各属性在z轴的层次关系：边界->背景颜色->背景图片->边框->填充->内容区。

从3D模型中，我们还可以得出以下结论。

（1）背景颜色、背景图片和边框之间是无法设置空白的。

（2）背景图片在背景颜色之上，也就是说背景图片可以覆盖背景色。

（3）元素背景指的是content和padding区域。

在CSS3中，它引进了border-shodaw，能把元素打扮得更加惊艳，轻易实现纸张卷边效果。如果元素的第一个元素为inset，那么它就是内阴影，否则为外阴影。此外CSS 2.1还有一个虚线框——outline。那么盒子模型的立体层次就变成：边界->虚线框->外阴影->背景颜色->背景图片->内阴影->影边框->填充->内容区，如图8-5所示。

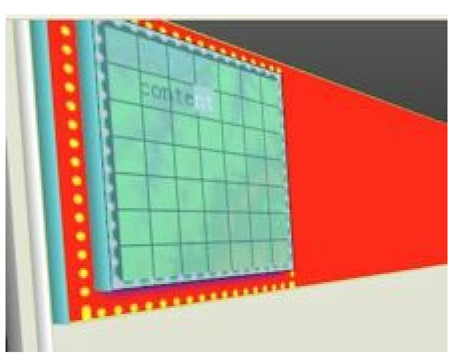


图8-5

在FF15下，从元素的3D图，可以直接看到它们之间的层叠关系。

不过真正影响布局的还是那几个，margin、border、padding、content。在早期，存在两种盒子模型，它决定着width、height的计算公式。W3C的盒子模型，亦即后来的content-box，内容区的宽即为el.style.width；IE在怪异模式的盒子模型，亦即后来的border-box，内容区的宽是未知的，要用width–左右边框宽–左右填空宽。这就导致IE的盒子总比W3C的smat。从设计与计算的角度，尤其是百分比设置宽高，IE的盒子模型更为合理，符合人们的常识。因此W3C后来搞了个box-sizing的CSS3新属性来搪塞人们的疑问。box-sizing在W3C规范中，拥有3种值，content-box、padding-box、border-box，宽高的计算起点由它们的名字决定。

8.3.6　元素的尺寸

由于元素的高与宽的取法都是一样的，笔者这里只拿宽来讲解。参照jQuery的行为，width是指内容区的宽。一般情况下，我们可以使用getComputedStyle精确得到元素的宽，但如果元素的display为none，或元素的祖先的dispaly为none，又或者元素脱离了DOM树，getComputedStyle就无能为力了。旧版本IE那边也差不多。并且浏览器在非content-box模式下，el.currentSyle.width或getComputedStyle(el, null).width得到的值是整个盒子的宽，不是内容区的宽（FF好像是例外），因此我们需要另辟蹊径。

首先要把隐藏元素显示出来。如果元素是隐藏的，它的offsetWidth为0，但这不能作为判定元素是隐藏的充分条件，因为用户可能直接设置width：0px。这时，我们需要判定它的display值是否为none。

```
function showHidden(node, array) {
    //http://www.cnblogs.com/rubylouvre/archive/2012/10/27/2742
529.html
    if (node.offsetWidth <= 0) { //opera.offsetWidth可能小于0
        if (rdisplayswap.test(cssHooks['@:get'](node, 'display'
))) {
            var obj = {
                node: node
            }
            for (var name in cssShow) {
                obj[name] = node.style[name]
                node.style[name] = cssShow[name]
            }
            array.push(obj)
        }
        var parent = node.parentNode
        if (parent && parent.nodeType === 1) {
            showHidden(parent, array)
        }
    }
}
```

接下来，我们还要判定元素的盒子模型。IE在怪异模式下，盒子模式为border-box，如果不支持box-sizing，那么大家都是content-box，否则根据box-sizing值进入裁剪。裁剪什么呢？裁剪IE带来的好东西offsetWidth，现在它已被标准浏览器良好支持，并列入规范。

我们先做一个实验：

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>contentbox by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; cha
rset=UTF-8">
        <style type="text/css">
            body,html{
                height:100%;
                background:gray;
            }
            #parent {
                background: red;
                width:300px;
                height:300px;
                border:1px solid greenyellow;
            }
            #son{
                width:100px;
                height:100px;
                background-color:blue;
                padding:20px;
                margin:15px;
                overflow:hidden;
                border:5px solid yellow;
            }
        </style>
        <script>
            window.onload = function() {
                var el = document.getElementById("son");
                console.log(el.offsetWidth);
                console.log(window.getComputedStyle(el, null).w
idth);
            }
        </script>
    </head>
    <body>
        <div id="parent">
            <div id="son"> </div>
        </div>
    </body>
</html>
```

我们可以在firebug中看到son元素的盒子模型的具体参数与结构，如图8-6所示。

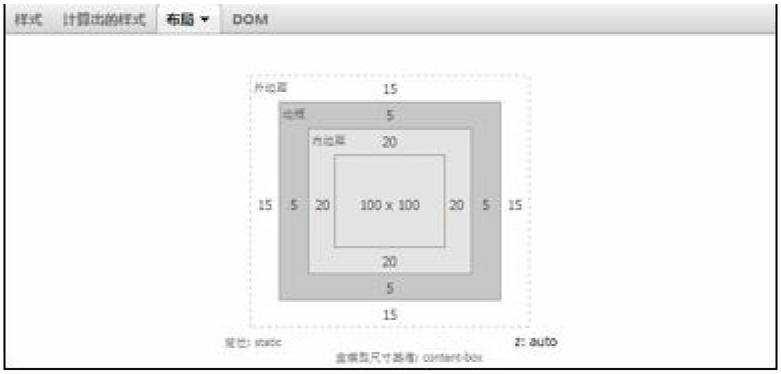


图8-6

我们也可以在IE8+的开发人员工具中看到类似的信息，如图8-7所示。

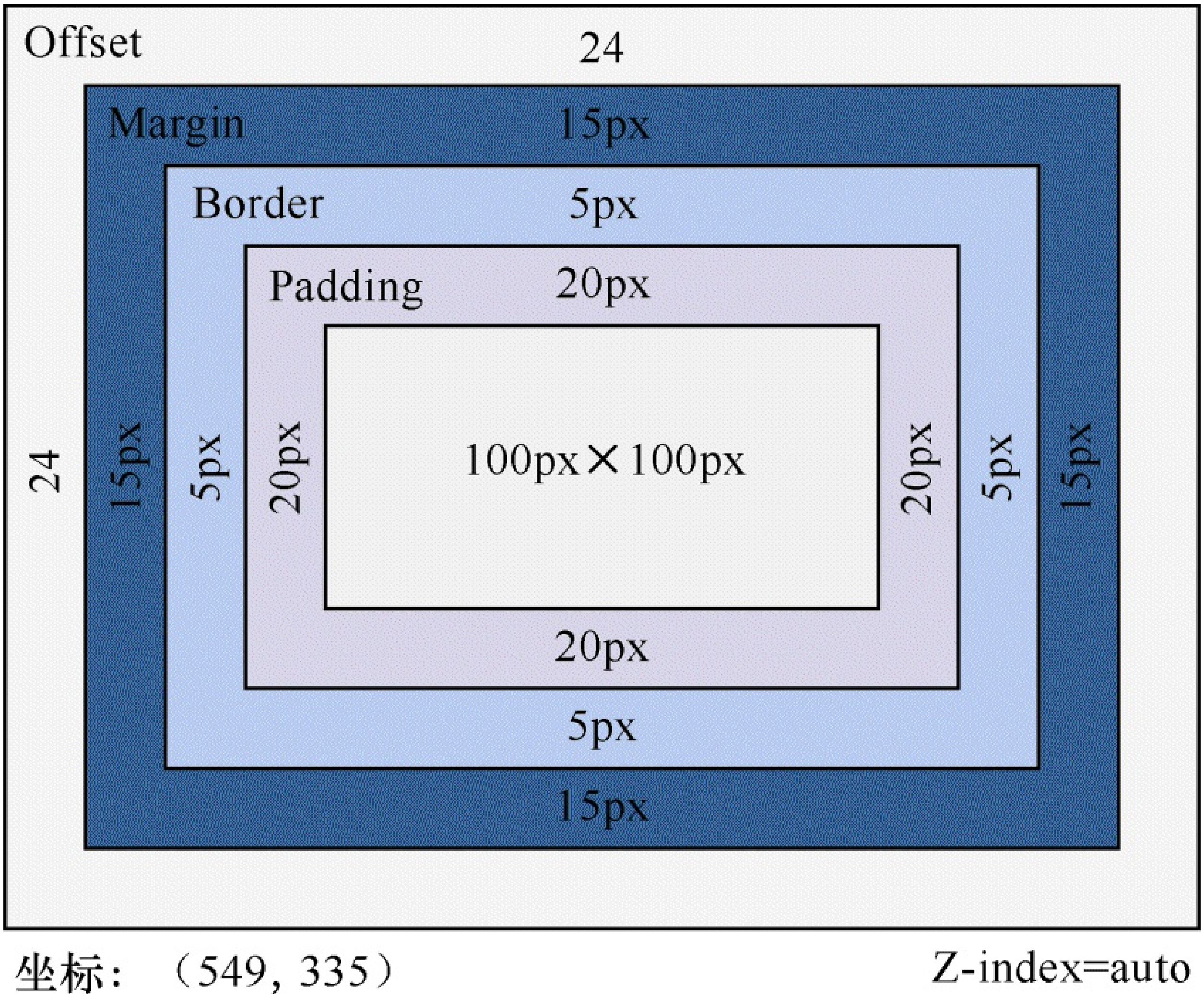


图8-7

控制台输出为150px，100px，我们可以从出推导出这样一个公式：

offsetWidth = borderWidth + paddingWidth + width

如果我们在div#son的样式添加多一行样式规则：

```
box-sizing:border-box;
-moz--sizing:border-box;
```

这时，红色方块就会比刚才的显示得小很多（Firefox在15版中依然显示错误）。控制台输出（100px，100px）（Firefox15为（100px，50px）），如图8-8所示。

offsetWidth = borderWidth + paddingWidth + width，这公式依然有效。

在笔者写本节时，只有Firefox支持padding-box。修改上面的实验代码，在firebug中看到控制台输出为110px，60px，如图8-9所示。

因此取元素内容的宽使用上面公式最合算不过，完全不受盒子模型影响。在jQuery的

dimensions模块中，又提供两个只读方法innerWidth、outerWidth用于取得padding-box、border-box与margin-box的宽。不过，依赖outerWidth进行加减，一点难度也没有。

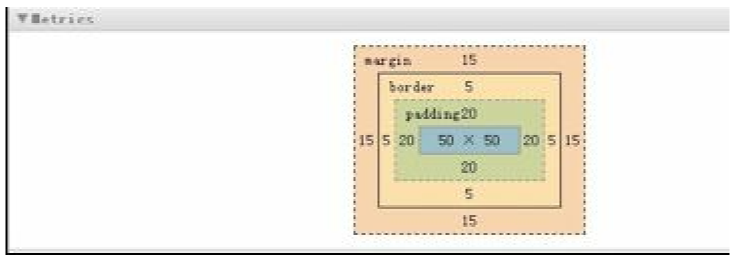


图8-8

不过设置元素内容区的宽就麻烦了，在非content-box中，它完全是一个间接值，一个不存在的属性。如果是padding-box，我们要将用户传入值加上padddingWidth；如果是border-box，再在刚才的基础上加上borderWidth。最后，在我们日常工作中，还有两个尺寸非常重要——窗口的大小与页面的大小，它们都是只读。为了方便起见，我们都把它们交由本模块处理。

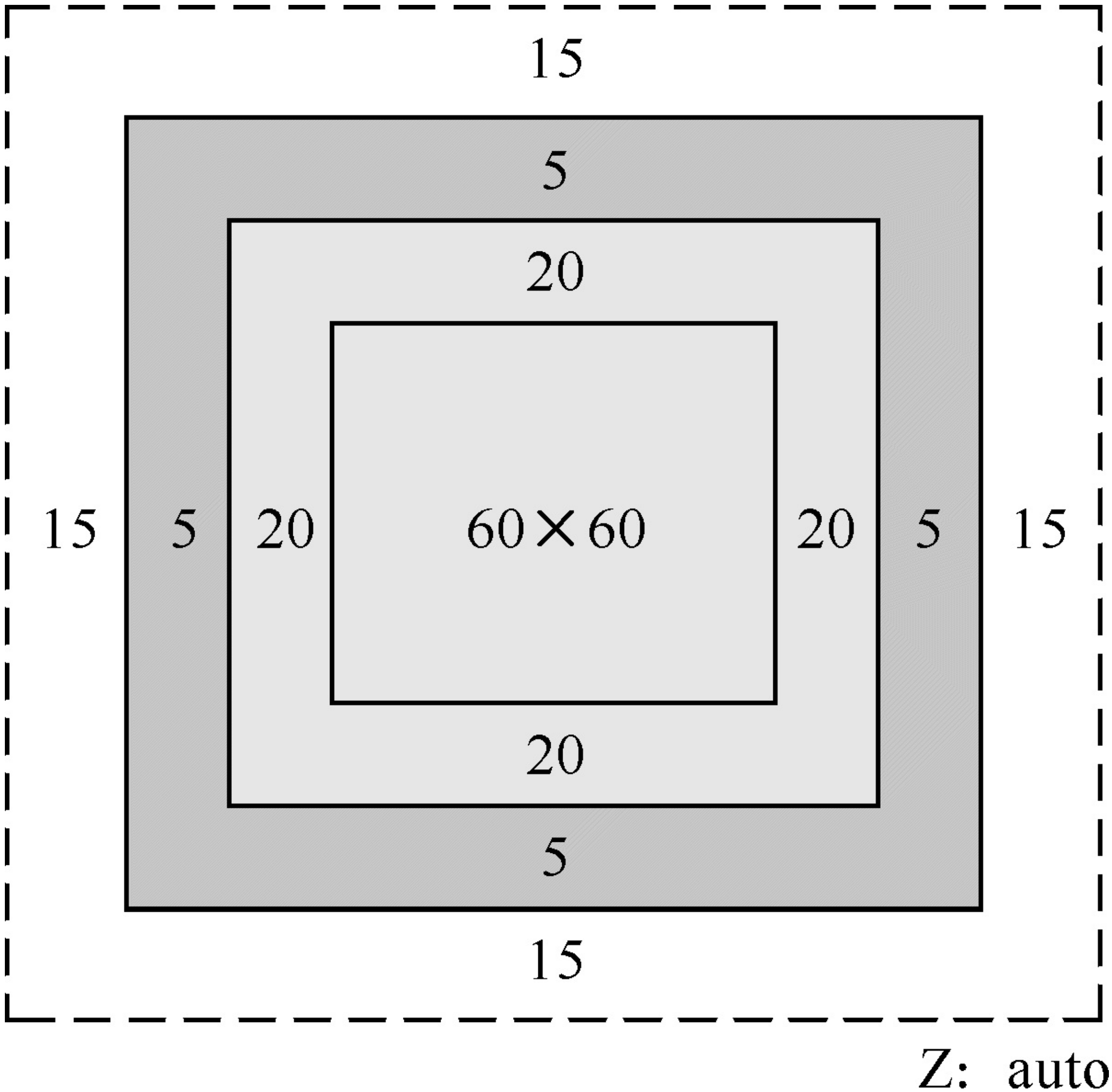


图8-9

先看窗口的宽。以前网景浏览器就提供了一个好用的只读属性innerWidth，标准浏览器一脉相承，都有这东西，IE9也支持它，W3C草稿已在

2011年收录它。因此如果你的框架只支持IE9+与较新版本的标准浏览器，只用它取窗口宽。旧版本IE，我们可以要区分一下它是否处于怪异模式。

怪异模式实质上就是IE6之前的模式。除了盒子模型的计算范围不一样外，另一个重要区别在于body标签的解释。在远古时代，body元素是最顶层的可视元素，而HTML元素保持隐藏。然后，现代浏览器认为body只是一个普通的块状元素，HTML则是包含整个浏览器窗口的可视元素。

IE又发明一套叫clientXXX的属性，用于取得元素的可视区的尺寸，它是不包含滚动条以及被隐藏的部分。因此窗口的宽可以这样取：

```
windowWidth = window.innerWidth || document.documentElement.clientWidth || document. body.clientWidth
```

不过，实际上jQuery与avalon等框架都不打算

支持怪异模式。怪异模式涉及许多东西，还有许多不可预测的怪异行为，而且殃及页面所有元素的盒子模型，不像box-sizing那样可以对单个元素进行设置。因此，上面的代码可以简化成这样：

```
windowWidth = document.documentElement.clientWidth
```

如果你的框架是手机框架，可以放胆去用innerWidth，即：

```
windowWidth = window.innerWidth
```

再看页面的宽，即文档的宽。一旦出现横向滚动条，我们就要考虑加上被隐藏的部分。标准浏览器又有一套叫outerXXX的属性，但那是取浏览器的尺寸的，因此不能像innerWidth那样照样画葫芦。IE在此又亲切地为我们奉上另两套叫scrollXXX、offsetXXX的属性，标准浏览器继续照抄不误，但

又抄得不好，留下兼容性问题。

offsetWidth

IE、Opera认为offsetWidth = clientWidth +滚动条+边框。

NS、FF认为offsetWidth是网页内容实际宽度，可以小于clientWidth。

scrollWidth

IE、Opera认为scrollWidth是网页内容实际宽度，可以小于clientWidth。

NS、FF认为scrollWidth是网页内容宽度，不过最小值是clientWidth。

你想想世界上有这么浏览器，谁知道它们又是怎么想的。因此直接把这些属性放在一起，取最大值吧。

```
var pageWidth = Math.max( document.documentElement.scrollWidth,

document.documentElement.offsetWidth, document.documentElement.
clientWidth,
document.body.scrollWidth, document.body.offsetWidth);
```

document.body.clientWidth肯定是最小的，不用比较。

在Firefox中还提供了window.scrollMaxX这样一个属性，它与window.innerWidth相加，恰好等于页面的宽度。不过MDC警告说它们之和不等于页面宽，可能是之前版本的问题吧。

在webkit系浏览器，它们直接在document中添加width/height属性，很方便。

```
var cssShow = {
    position: 'absolute',
    visibility: 'hidden',
    display: 'block'
}

var rdisplayswap = /^(none|table(?!-c[ea]).+)/

function showHidden(node, array) {
//...
}
```

```
avalon.each({
    Width: 'width',
    Height: 'height'
}, function (name, method) {
    var clientProp = 'client' + name,
            scrollProp = 'scroll' + name,
            offsetProp = 'offset' + name
    cssHooks[method + ':get'] = function (node, which, override
) {
        var boxSizing = -4
        if (typeof override === 'number') {
            boxSizing = override
        }
        which = name === 'Width' ? ['Left', 'Right'] : ['Top',
'Bottom']
        var ret = node[offsetProp] // border-box 0
        if (boxSizing === 2) { // margin-box 2
            return ret + avalon.css(node, 'margin' + which[0],
true) + avalon.css(node,
            'margin' + which[1], true)
        }
        if (boxSizing < 0) { // padding-box  -2
            ret = ret - avalon.css(node, 'border' + which[0] +
'
            Width', true) - avalon.css(node, 'border' + which[1
] + 'Width', true)
        }
        if (boxSizing === -4) { // content-box -4
            ret = ret - avalon.css(node, 'padding' + which[0],
t
            rue) - avalon.css(node, 'padding' + which[1], true)
        }
        return ret
    }
    cssHooks[method + '&get'] = function (node) {
        var hidden = [];
        showHidden(node, hidden);
        var val = cssHooks[method + ':get'](node)
        for (var i = 0, obj; obj = hidden[i++]; ) {
            node = obj.node
            for (var n in obj) {
                if (typeof obj[n] === 'string') {
                    node.style[n] = obj[n]
                }
            }
```

```
        }
        return val
    }
    avalon.fn[method] = function (value) { //会忽视其display
        var node = this[0]
        if (arguments.length === 0) {
            if (node.setTimeout) { //取得窗口尺寸
                return node['inner' + name] ||
                        node.document.documentElement[clientPro
p] ||
                        node.document.body[clientProp]
                        //IE6下前两个分别为undefined,0
           }
           if (node.nodeType === 9) { //取得页面尺寸
                var doc = node.documentElement
                //FF chrome    html.scrollHeight< body.scrollHe
ight
                //IE 标准模式 : html.scrollHeight> body.scrollHei
ght
                //IE 怪异模式 : html.scrollHeight 最大等于可视窗口多
一点?
                return Math.max(node.body[scrollProp], doc[scro
ll
                lProp], node.body[offsetProp], doc[offsetProp],
 doc[clientProp])
           }
           return cssHooks[method + '&get'](node)
        } else {
           return this.css(method, value)
        }
    }
    avalon.fn['inner' + name] = function () {
        return cssHooks[method + ':get'](this[0], void 0, -2)
    }
    avalon.fn['outer' + name] = function (includeMargin) {
        return cssHooks[method + ':get'](this[0], void 0, inclu
de
        eMargin === true ? 2 : 0)
    }
})
```

8.3.7　元素的显隐

想让元素在页面上看不见有许多办法，但我这里只讲display。display为none时，它不占有物理空间，附近的元素就可以顺势向它的位置挪过去，比如手风琴效果，下拉效果都依赖于它。因此这种隐藏方式非常有用。在许多情况下，它是没有什么好说的，显示就设置block，隐藏就是none，切换就是根据它原来的状态决定显隐。

但不是所有元素直接用block就能搞定，像thead、tbody、tr等具有特定默认display值的元素，它们一旦设置了block，表格就会面目全非。

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>display by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; cha
rset=UTF-8">
        <style type="text/css">
            body,html{
                height:100%;
                background:gray;
            }
            #parent {
                background: red;
```

```
                width:300px;
                height:300px;
            }
            #son{
                width:100px;
                height:100px;
                background:blue;
                padding:20px;
                margin:15px;
                display:none;
                border:5px solid yellow;
            }
        </style>
        <script>
            window.onload = function(){
                var el = document.getElementById("son")
                el.style.display= ""

            }
        </script>
    </head>
    <body>
        <div id="parent">
            <div id="son"> </div>
        </div>
    </body>
</html>
```

我们必须赋以正确的display值才能让它生效。问题是，如何界定这个“正确”。单单是块状元素设置block，内联元素设置inline，让它们显示是不行的。这还引进了一个问题，如何区分它是块状元素或内联元素。对于不断增长的HTML5元素种类，

hash法也不可靠。而且增长的不单是元素种类，还有display值的类型。

在老旧的CSS 1规范中，display值仅包括：block、inline、list-item、none。在CSS 2.1规范中，它已经扩张到如下这么多值：inline、block、list-item、run-in、inline-block、table、inline-table、table-row-group、table-header-group、table-footer-group、table-row、table-column-group、table-column、table-cell、table-caption、none、inherit。CSS3引入更强大的布局模型Flexible Box，对display增加ruby、ruby-base、ruby-text、ruby-base-group、ruby-text-group、flex、grid等值（注，新添值不太稳定，一切以W3C上的最新草稿为准）。

不过有些元素，我们可以肯定它们的默认样式值，对于一些特别的元素就需要实时去取了。为了取得干净的默认display属性，我们需要在iframe沙

箱中去取。

```
var none = 'none'
function parseDisplay(elem, val) {
    //用于取得此类标签的默认display值
    var doc = elem.ownerDocument
    var nodeName = elem.nodeName
    var key = '_' + nodeName
    if (!parseDisplay[key]) {
        var temp = doc.body.appendChild(doc.createElement(nodeN
ame))
        if (avalon.modern) {
            val = getComputedStyle(temp, null).display
        } else {
            val = temp.currentStyle.display
        }
        doc.body.removeChild(temp)
        if (val === none) {
            val = 'block'
        }
        parseDisplay[key] = val
    }
    return parseDisplay[key]
}
```

我们通过toggle方法设置元素是否显示或隐藏。

```
function toggle(node, show){
        var display = node.style.display, value
        if (show) {
            if (display === none) {
                if (!value) {
                    node.style.display = ''
                }
            }
```

```
            if (node.style.display === '' && avalon(node).css('
display') === none &&
                    // fix firefox bug,必须挂到页面上
                    avalon.contains(node.ownerDocument, node))
{

                value = parseDisplay(node)
            }
        } else {
            if (display !== none) {
                value = none
            }
        }
        if (value !== void 0) {
            node.style.display = value
        }
}
```

8.3.8　元素的坐标

元素的坐标就是指其top与left值。node.style恰逢有这两个属性，但它只有被定位了才有效，否则即使在IE9、FF15、Chrome23、Opera12等浏览器下都返回auto。幸好，即使一个元素没有被定位，它的offsetTop、offsetLeft也是有效的，它们是相对于offsetParent的距离。我们一级级向上累加，就能得到相对页面的坐标，亦有人称之为元素的绝对坐标。

```
function offset(node) {
    var left = node.offsetLeft,
            top = node.offsetTop;
    do {
        left += node.offsetLeft;
        top += node.offsetTop;
    } while (node = node.offsetParent);
    return {
        left: left,
        top: top
    }
}
```

此外，相对于可视区的坐标也很实用，比如让弹出窗口居中对齐。以前实现这个计算量非常巨大，自从IE的getBoundingClientRect方法被发掘出来后，简直是小菜一碟。更令人高兴的是，它现在也被标准浏览器普遍接受，并列入W3C标准，无兼容性之忧。此方法能获取页面中某个元素（border-box）的左、上、右和下分别相对浏览器视窗的位置。它返回一个Object对象，该对象肯定有这样4个属性：top、left、right、bottom，标准浏览器下可能还多出width、height这两个属性。这里的top、left和CSS中的理解很相似，width、height是元素自

身的宽高。但是right、bottom和CSS中的理解有点不一样，right是指元素右边界距窗口最左边的距离，bottom是指元素下边界距窗口最上面的距离，如图8-10和图8-11所示。

微软的MSDN《Measuring Element Dimension and Location with CSSOM in Internet Explorer 9》给出另一个更具体的图，标著更多CSSOM各种属性。示例页面中拥有一个相对定位的红色元素。蓝色元素是红色元素的父节点，它是用于演示各种盒子，如content-box、padding-box、border-box、margin-box以及offsetTop（这是属于蓝色元素的，换言之，它是蓝色元素的offsetParent）。viewport是个黑色虚框，为html标签。蓝色元素还拥有滚动条，方便我们观察clientTop与scrollTop或者clientHeight与scrollHeight的差异。getBoundingClientRec中的top与bottom也在图8-12中显示出来。

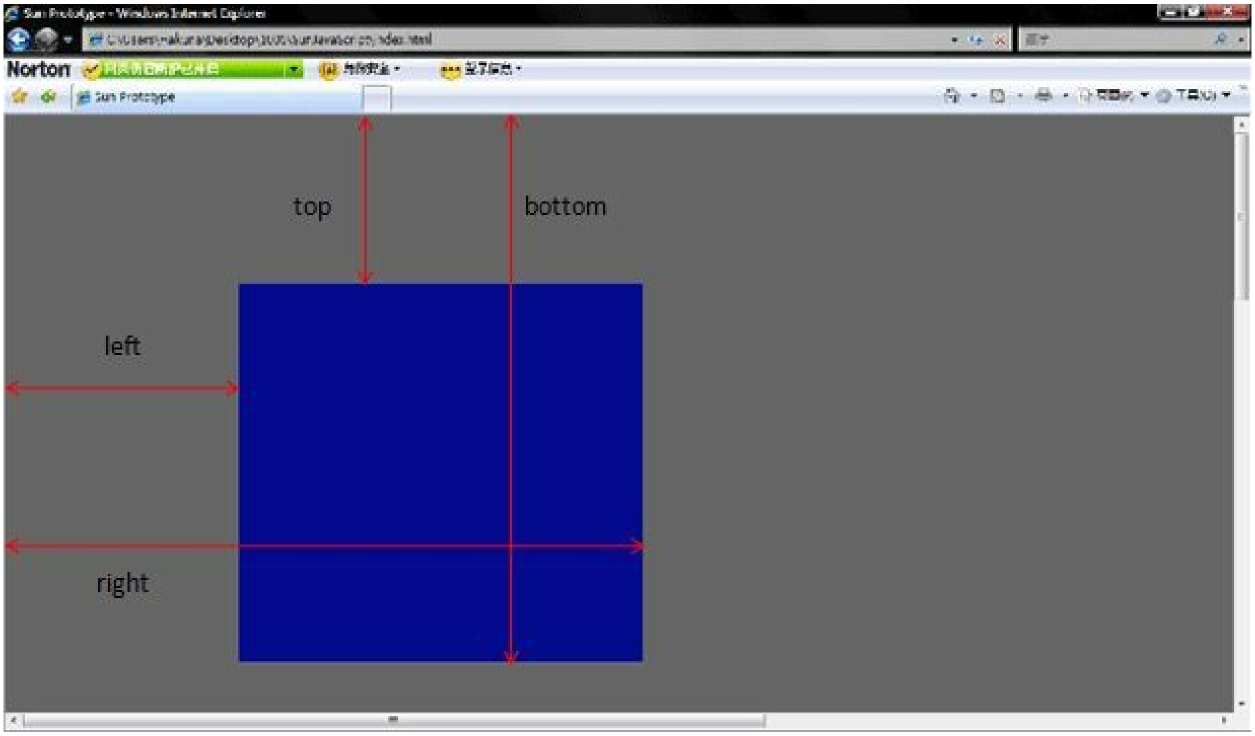


图8-10

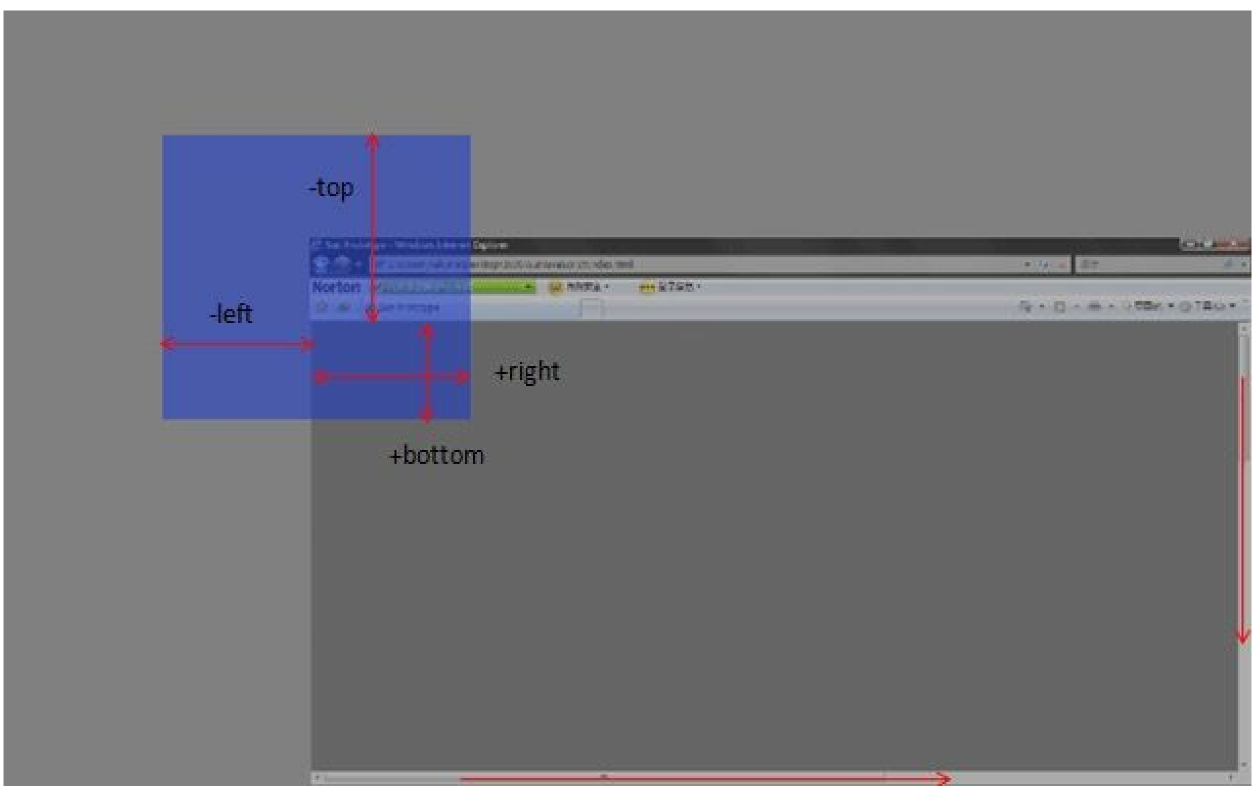


图8-11

```
http://msdn.microsoft.com/en-us/library/ms530302%28VS.85%29.aspx
```

getBoundingClientRect的支持情况如表8-2所示。

表**8-2**

浏览器	**Chrome**	**Firefox (Gecko)**	**IE**	**Opera**	**Safari**
支持	1.0	3.0 (1.9)	4.0	(Yes)	4.0

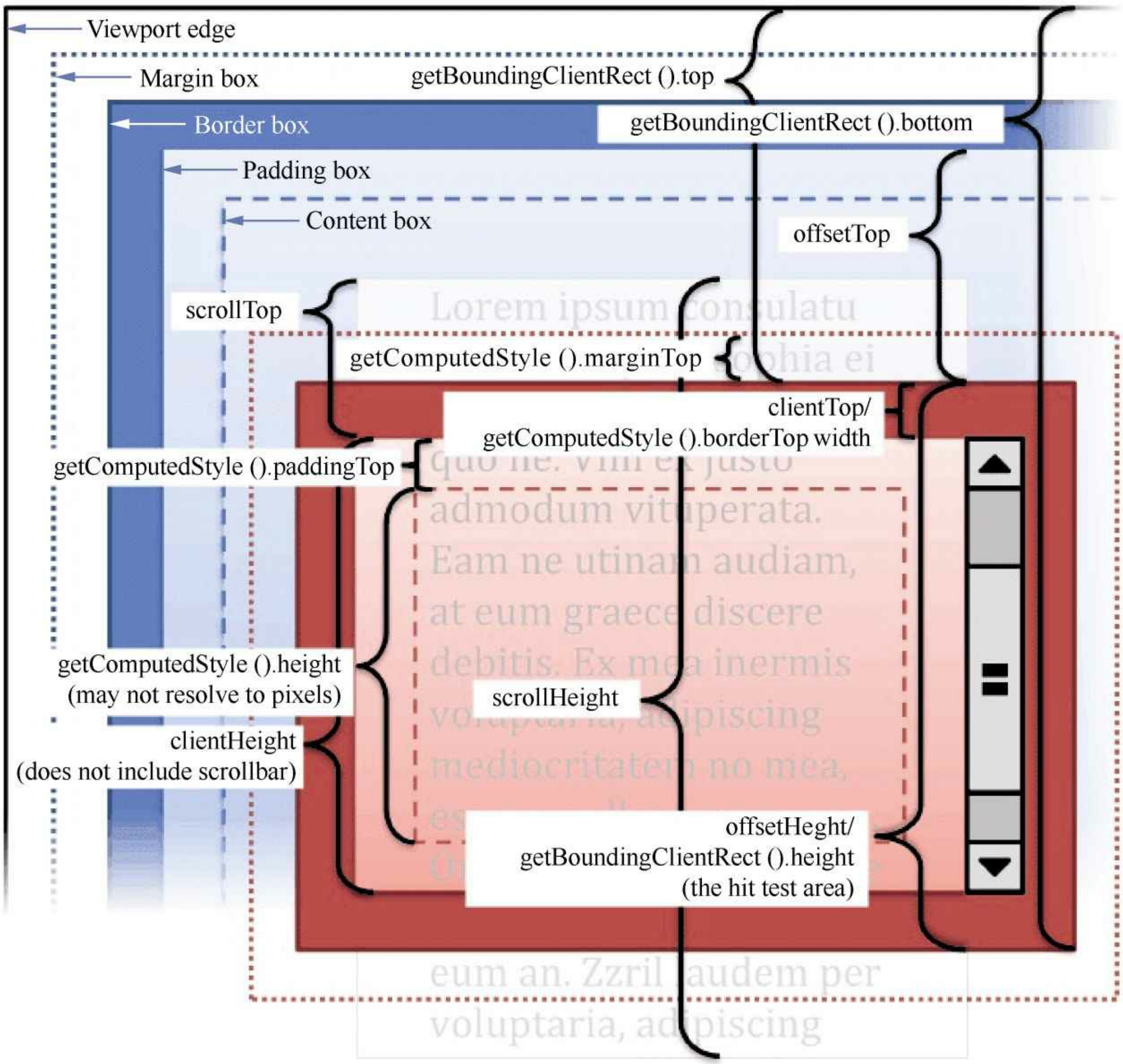


图8-12

因此我们也可以很方便地利用它求出相对于页面的距离，将它相对于可视区的距离加上滚动距离就行了。

```
var rect =this.getBoundingClientRect()
var top = rect.top+document.documentElement.scrollTop;
var left = rect.left+document.documentElement.scrollLeft;
```

到目前为止，我们只是把所有用到的知识介绍一遍，就像W3C文案那样总是完美而不实用，我们还要考虑浏览器兼容性呢。有了兼容性，上面简洁的公式就要改得非常臃肿丑鄙才可以用。

那真正的解法是如何的呢？首先，我们判定它是否在DOM树上，不在直接返回（0，0）。否则取得元素在可视区的距离加上滚动距离然后减去浏览器的边框。因此，计算可视区距离与滚动距离时都已经包含浏览器的边框。而浏览器的边框即最顶层的可视元素的边框，在标准模式下，顶层可视元素为html，怪异模式下为body。计算滚动距离也是这样，需要选好顶层可视元素。由于Windows8与IE10的发布，加速了旧版本IE的淘汰，像jQuery与avalon已经不考虑支持怪异模式。各位可以参考自

己公司的客户组成斟酌一下是否支持怪异模式。

```
avalon.fn.offset = function () { //取得距离页面左右角的坐标
    var node = this[0],
            box = {
                left: 0,
                top: 0
            }
    if (!node || !node.tagName || !node.ownerDocument) {
        return box
    }
    var doc = node.ownerDocument,
            body = doc.body,
            root = doc.documentElement,
            win = doc.defaultView || doc.parentWindow
    if (!avalon.contains(root, node)) {
        return box
    }
    //http://hkom.blog1.fc2.com/?mode=m&no=750 body的偏移量是不包
含margin的
    //我们可以通过getBoundingClientRect来获得元素相对于client的rect.
    //http://msdn.microsoft.com/en-us/library/ms536433.aspx
    if (node.getBoundingClientRect) {
        box = node.getBoundingClientRect() // BlackBerry 5, iOS
 3 (original iPhone)
    }
    //chrome/IE6: body.scrollTop, firefox/other: root.scrollTop
    var clientTop = root.clientTop || body.clientTop,
       clientLeft = root.clientLeft || body.clientLeft,
       scrollTop = Math.max(win.pageYOffset || 0, root.scrollTo
p, body.scrollTop),
       scrollLeft = Math.max(win.pageXOffset || 0, root.scrollL
eft, body.scrollLeft)
    // 把滚动距离加到left,top中去。
    // IE一些版本中会自动为HTML元素加上2px的border，我们需要去掉它
    // http://msdn.microsoft.com/en-us/library/ms533564(VS.85).
aspx
    return {
        top: box.top + scrollTop - clientTop,
        left: box.left + scrollLeft - clientLeft
    }
}
```

我们再来取元素相对于其offsetParent的位置，亦有人称之为元素的相对坐标。要取得此值，我们先要确定其offsetParent。根据W3C给出的规律，元素是这样寻找其offsetParent的。如果元素被移出DOM树或display，为none；作为HTML或BODY元素，或其position的精确值为fixed时，返回null。否则分两种情况，当position为absolute、relative的元素的offsetParent时，它总是为其最近的已定位的祖先，没有找最近的td，th元素，再没有返回body；当position为static的元素的offsetParent时，则是先找最近的td、th、table元素，再没有返回body。但现实中，Firefox在position为fixed返回body；在IE6～IE8下，会增加一条规则，先寻找离元素最近的设置有能激活hasLayout的祖先元素。

假若依据这个规律，我们在太多情况下会得到offsetParent为null，导致无法计算。我们看jQuery是

怎么做的。jQuery也有个offsetParent方法，它是将选中元素的所有“offsetParent”收集起来，重新包装为jQuery对象返回。而这个“offsetParent”的定义被它修改了。浏览器认为offsetParent最高只能取到body，而且存在为null的情况；jQuery则认为元素的offsetParent的position必须为relative或absolute，否则继续回上寻找另一个被定位的祖先，没有返回html。另外，jQuery认为position:fixed的元素也有offsetParent，就是当前可视区。

为了让大家有个直观的认识，还是建一个HTML页面。

```
<!DOCTYPE HTML>
<html id="html">
    <head>
        <title>offsetParent by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; cha
rset=UTF-8">
        <style type="text/css">
            body,html{
                height:100%;
                background:orange;
            }
            body{
                margin:20px;
            }
```

```
            #parent {
                position: relative;
                width:250px;
                height:250px;
                margin:20px;
                background:aqua;
                border:20px solid red;
                padding:20px;
            }
            #son {
                position: absolute;
                width:150px;
                height:150px;
                margin:20px;
                background:fuchsia;
                border:20px solid blue;
                padding:2px;
            }
        </style>
    </head>
    <body id="body">
        <div id="parent">
            <div id="son"></div>
        </div>
    </body>
</html>
```

从盒子模型的角度来看，相对于offsetParent的距离，是指此元素的margin-box的左上角到offsetParent的content-box的左上角的距离。由于offsetParent、getBoundingClientRect等都是由IE首先提出来的，因此盒子都以border-box为计算单元，我们需要减少offsetParent的左边框与元素的左边界

的宽，如图8-13所示。

因此x轴距离的计算公式如下。

```
X = node[clientLeft] – offsetParent[client_left] -
offsetParent[borderLeftWidth] – node[marginLeftWidth]
```

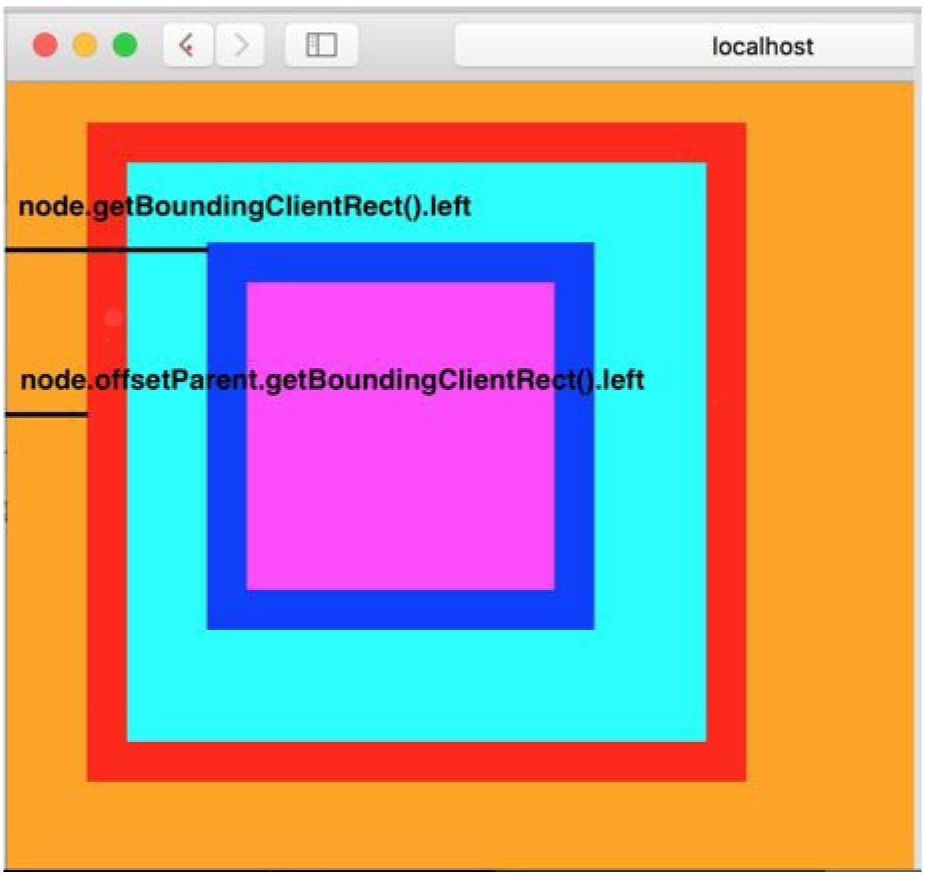


图8-13

整个方法实现如下。

```
avalon.fn.position = function () {
    var offsetParent, offset,
            elem = this[0],
            parentOffset = {
                top: 0,
                left: 0
            }
    if (!elem) {
        return parentOffset
    }
    if (this.css('position') === 'fixed') {
        offset = elem.getBoundingClientRect()
    } else {
        offsetParent = this.offsetParent() //得到真正的offsetPare
nt
        offset = this.offset() // 得到正确的offsetParent
        if (offsetParent[0].tagName !== 'HTML') {
            parentOffset = offsetParent.offset()
        }
        parentOffset.top += avalon.css(offsetParent[0], 'border
TopWidth', true)
        parentOffset.left += avalon.css(offsetParent[0], 'borde
rLeftWidth', true)

        // Subtract offsetParent scroll positions
        parentOffset.top -= offsetParent.scrollTop()
        parentOffset.left -= offsetParent.scrollLeft()
    }
    return {
        top: offset.top - parentOffset.top - avalon.css(elem, '
marginTop', true),
        left: offset.left - parentOffset.left - avalon.css(elem
, 'marginLeft', true)
    }
}
avalon.fn.offsetParent = function () {
    var offsetParent = this[0].offsetParent
    while (offsetParent && avalon.css(offsetParent, 'position')
 === 'static') {
        offsetParent = offsetParent.offsetParent;
    }
    return avalon(offsetParent || root)
}
```

有了position方法，我们就可以正确计算出top、left样式值了。我们继续在cssHooks对象添加钩子函数。

```
'top,left'.replace(avalon.rword, function (name) {
    cssHooks[name + ':get'] = function (node) {
        var computed = cssHooks['@:get'](node, name)
        return /px$/.test(computed) ? computed :
                avalon(node).position()[name] + 'px'
    }
})
```

8.4　元素的滚动条的坐标

这是浏览器一组非常重要的属性，因此光是浏览器本身就提供了多个方法来修改它们，比如挂在window下的scroll、scrollTo、scrollBy方法以及挂在元素下的scrollLeft、scrollTop、scrollIntoView方法。jQuery在css模块就提供了scrollLeft、scrollTop来修改或读取元素或窗口的滚动坐标，在animation模块，更是允许它以更平滑的方式来挪动它们。EXT更是用整一个模块来满足用户对滚动条的各种需求。

修改top、left来挪动元素有一个坏处，可能遮在某些元素之上，而修改scrollTop、scrollLeft不会。这里我们还是仿照jQuery那样，把这两个方法取名为scrollTop、scrollLeft。对于一般的元素节点，读写它们没有什么难点，因为元素上就有这两个属性了。我们只需集中精力对付最外面的滚动

条，也称浏览器的滚动条，位于最顶层的可视元素之上。设置时，我们要用到window中的scrollTo方法，里面传入你要滚到的坐标。读取时，我们尝试使用pageXOffset、pageYOffset这组属性，标准浏览器从网景时代就支持了，IE则直接取html元素的scrollLeft, scrollTop属性，因为笔者也不打算支持怪异模式。想支持怪异模式的朋友，可以继续尝试从body元素中取相关属性。

```
//生成avalon.fn.scrollLeft, avalon.fn.scrollTop方法
avalon.each({
    scrollLeft: 'pageXOffset',
    scrollTop: 'pageYOffset'
}, function (method, prop) {
    avalon.fn[method] = function (val) {
        var node = this[0] || {},
                win = getWindow(node),
                top = method === 'scrollTop'
        if (!arguments.length) {
            return win ? (prop in win) ? win[prop] : root[metho
d] : node[method]
        } else {
            if (win) {
                win.scrollTo(!top ? val : avalon(win).scrollLef
t(), top ? val : avalon(win).
                scrollTop())
            } else {
                node[method] = val
            }
        }
    }
})
```

```
function getWindow(node) {
    return node.window && node.document ? node : node.nodeType
=== 9 ? node.defaultView || node.parentWindow : false;
}
```

8.5 总结

本章通过一个强大的适配器对象（钩子对象），使得我们的 css 方法具有强大的扩展性，进而解决了各种样式的读写问题。

第9章　属性模块



本章所讲的属性在英文中分别称之为property与attribute。property是DOM对象自身就拥有的属性，而attribute是我们通过设置HTML标签而给之赋予的特性。笔者在stackoverflow上找到如下的回答，或者会更加接近于真正的答案：

http://stackoverflow.com/questions/258469/what-

is-the-difference-between-attribute-and-propertyThese words existed way before Computer Science came around.

Attribute is a quality or object that we attribute to someone or something . For example, the scepter is an attribute of power and statehood.

Property is a quality that exists without any attribution . For example, clay has adhesive qualities; or, one of the properties of metals is electrical conductivity. Properties demonstrate themselves though physical phenomena without the need attribute them to someone or something. By the same token, saying that someone has masculine attributes is self-evident. In effect, you could say that a property is owned by someone or something.

To be fair though, in Computer Science these two

words, at least for the most part, can be used interchangeably - but then again programmers usually don't hold degrees in English Literature and do not write or care much about grammar books :).

最关键的两句话。

（1）attribute（特性），是我们赋予某个事物的特质或对象。

（2）property（属性），是早已存在的不需要外界赋予的特质。

但这还是比较绕口，因此本书通过加点定语来解决它们，也切合本章的标题。attribute就叫自定义属性，property则叫固有属性。

9.1 元素节点的属性

通常我们把对象的非函数成员叫做属性。对于元素节点来说，其属性大体分成两大类，固有属性与自定义属性（特性）。固有属性一般遵循驼峰命名风格，拥有默认值，并且无法删除。自定义属性是用户随意添加的键值对，由于元素节点也是一个普通的JavaScript对象，没有什么严格的访问操作，因此命名风格林林总总，值的类型也乱七八糟。但是随意添加属性显然不够安全，比如引起循环引用什么的。因此，浏览器提供了一组API来供人们操作自定义属性，即setAttribute、getAttribute、removeAttribute。当然还有其他API，不过这是标准套装，只有在IE6、IE7那样糟糕的环境下，我们才求助于其他API，一般情况下这3个足矣。我们通称它们为DOM属性系统。DOM属性系统对属性名会进行小写化处理，属性值会统一转字符串。

```
var el =  document.createElement("div")
el.setAttribute("xxx", "1")
el.setAttribute("XxX", "2")
el.setAttribute("XXx", "3")
console.log(el.getAttribute("xxx"))
console.log(el.getAttribute("XxX"))
```

IE6、IE7会返回“1”，其他浏览器返回“3”。在前端的世界，我们真是走到哪都能碰到兼容性问题。这只是冰山一角，IE6、IE7在处理固有属性时要求进行名字映射，比如class变成className、for变成 htmlFor。对于布尔属性（一些只返回布尔的属性），浏览器间的差异更大，具体如表 9-1所示。

```
<input type="radio"  id="aaa">
<input type="radio" checked id="bbb">
<input type="radio" checked="checked" id="ccc">
<input type="radio" checked="true" id="ddd">
<input type="radio" checked="xxx" id="eee">

"aaa,bbb,ccc,ddd,eee".replace(/\w+/g,function( id ){
    var elem =  document.getElementById( id )
    console.log(elem.getAttribute("checked"));
})
```

表**9-1**

	#aaa	#bbb	#ccc	#ddd	#eee
IE7	false	true	true	true	true
IE8	""	"checked"	"checked"	"checked"	"checked"
IE9	null	""	"checked"	"true"	"xxx"
FF15	null	""	"checked"	"true"	"xxx"
Chrome23	null	""	"checked"	"true"	"xxx"

因此框架很有必要提供一些API来屏蔽这些差异性。但在IE6统治时期，这个需求并不明显，因为IE6、IE7不区分固有属性与自定义属性。setAttribute与getAttribute在当时的人看来只是一个语法糖，用`el.setAttribute("innerHTML","xxxx")`与用`el.innerHTML = "xxx"`效果一样，而且后者更方便。即使早期应用那么广泛的Prototype.js，提供属性操作API也非常贫乏，只有identify、readAttribute、writeAttribute、hasAttribute、

classNames、hasClassName、addClassName、removeClassName、toggleClassName与$F方法。Prototype.js也察觉到固有属性与自定义属性在DOM属性系统的差异，在它的内部，搞了个`Element._attributeTranslations`。然而，Prototype.js这个属性系统内部还是优先使用el[name]方式来操作属性，而不是set/getAttribute（接下来的章节，笔者会分析它是如何实现的）。

jQuery早期的attr方法，其行为与Prototype一模一样。只不过jQuery1.6之前，是使用attr方法同时实现了读、写、删掉这3种操作。从易用性来说，不区分固有属性与自定义属性，由框架自动内部处理应该比attr、prop分家更容易接受。那么是什么逼迫jQuery这样做的呢？这是因为选择器引擎。

jQuery是最早以选择器为向导的类库。它最开始的选择器引擎是Xpath式，后来换成Sizzle，以

CSS表达式风格来选取元素。在CSS2.1中引入了属性选择器[aaa=bbb]，IE7也开始残缺支持。Sizzle当然毫不含糊地实现了这语法。属性选择器是最早突破类名与ID的限制求取元素的。为了显摆它的强大，设计者让它拥有多种形态，满足人们各种匹配需要。比如，它可以只写属性名[checked]，那么上例中的后四个元素都选中。[checked=true]选中第四个元素，[checked=xxx]选中第五个元素，其中true与xxx都是用户在标签的预设值，而一字不差地取回这个预设值的工作也只有getAttribute才能做到。根据标准，setAttribute是应该返回用户设置的字符串。而el[xxx]这样的取法就不一定了，比如el.checked就返回布尔值，表示两种状态。这种通过状态取元素的方式就不归属性选择器管了，CSS3又新设了个状态伪类满足人们的需求。

此外，属性还能以`[name^=value]`、`[name*=value]`、`[name$=value]`等更精致的方

式来甄选元素，而这一切都建立在获取用户预设值的基础上。因此jQuery下了很大决心，把prop从attr切割出来。虽然为了满足用户的向前兼容需求，又“偷偷”地让attr做了prop的事，但以此为契机，jQuery团队挖掘出更多兼容性问题与相应解决方案。元素内部撑起整个属性系统的attributes类数组属性也从幕后走到前台，为世人所知。浏览器经过这么多年的发展，谁也说不清某个元素节点拥有多少个属性。for..in循环也不行，因为它对不可遍历的属性无能为力。在IE6、IE7中，attributes会包含上百个特性节点，不管你是用setAttribute定义的属性，还是以el[xxx]=yyy的定义的属性，还是没有定义的属性。可惜到IE8与其他浏览器中，你只看到寥寥可数的几个特性节点，称为显式属性（specified attribute）。

显式属性就是被显式设置的属性，分两种情况，一种是写在标签内的HTML属性，一种是通过

setAttribute动态设置的属性。这些属性不分固有还是自定义，只要设置了，就出现在attributes中。在IE6、IE7中，我们也可以通过特性节点的specified属性判定它是否被显式设置了。在IE8或其他浏览器中，我们想判定一个属性是否为显式属性，可以直接用hasAttribute API判定。

```
var  isSpecified = !"1"[0] ? function(el, attr){
      return el.hasAttribute(attr)
} : function(el, attr){
      var val = el.attributes[attr]
      return !!val && val.specified
}
```

此外，HTML5对属性进行了更多的分类，打包到不同的对象中去。dataset对象装载着所有以data-开头的自定义属性。classList装载着元素的所有类名，并且提供一套API来操作它们。formData装载着所有要提供到后台的数据，以表单元素的name值与value值构成的不透明对象。尽管如此，还是有大量属性是没有编制的，它们代表着元素的

各种状态以及与其他元素的联结。正因为如此，它们的值才五花八门，如uniqueNumber、tabIndex、colspan、rowspan为整数，designMode、unselectable、autocomplete的值不是on就是off，iframe通过frameborder的值（是0还是1）决定是否显示边框，通过scrolling的值（是yes还是no）决定是否显示滚动条，表单元素的form属性总是指向其外围的表单对象，表单元素的checked、disabled、readOnly等属性总是返回布尔值……

面对如此庞杂的属性，主流框架也纷纷建立了对应的模块来整治它。在jQuery中有attributes模块，YUI3是dom-class、dom-attrs模块，dojo是dom-attr、dom-prop、dom-class模块。从内容来看，类名都被单独提出来处理，在jQuery中，表单元素的value也被单独提出来处理。Prototype.js虽然没有划分出来，但对付一般属性有readAttribute与writeAttribute，ID有identify，表单元素的value有

$F，类名更是对应多个方法，这个阵营与jQuery的属性模块一模一样。在1.5版之前，Prototype.js的属性模块一直是jQuery属性模块的蓝本。

9.2　如何区分固有属性与自定义属性

在jQuery、mass Framework中，提供两组方法来处理它们。但用户首先要知道他正在处理的东西是何物。虽然我们可以在以下链接查到每个元素拥有什么方法与属性，但显然不是每个人都那么勤奋。

```
http://msdn.microsoft.com/library/ms533029%28v=VS.85%29.aspx
```

我们需要做些实验找出其规律，从这些规律中，我们可以找到区分固有属性与自定义属性的方案。

```
 <!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta content="IE=8" http-equiv="X-UA-Compatible"/>
    <title>如何区分属性与特性 by  司徒正美</title>
    <script type="text/javascript">
      var test = function(){
        var a = document.getElementById("test");
        a.setAttribute("title", "title");
```

```
        a.setAttribute("title2", "title2");
        alert(a.parentNode.innerHTML);
      }
    </script>
  </head>
  <body>
    <p><strong id="test">司徒正美</strong></p>
    <p><button type="button" onclick="test()">点我，进行测试</butt
on></p>
  </body>
</html>
```

IE8下打印出：

```
<STRONG id=test title=title title2="title2">司徒正美</STRONG>
```

Firefox15、Chrome23、IE9下打印出：

```
<strong title2="title2" title="title" id="test">司徒正美</strong>
```

再将a.setAttribute("title", "title"); a.setAttribute("title2", "title2")这两行改成a.title = "title";a.title2 = "title2"。

IE8下打印出：

```
<STRONG id=test title=title title2="title2">司徒正美</STRONG>
```

Firefox15、Chrome23、IE9下打印出：

```
<strong title2="title2" title="title" id="test">司徒正美</strong>
```

经过观察，旧版本IE下固有属性，如title、id是不带引号的，自定义属性是带引号的，但在标准浏览器下，我们无法区分，否决此方案。

在元素中有一个attributes属性，里面有许多特性节点，这些特性节点在IE下有一种名为expando的布尔属性，可以判定它是否为自定义属性。但在标准浏览器下没有此属性，否决此方案。

```
function isAttribute(attr, host){ //仅有IE
   var attrs =  host.attributes;
   return attrs[attr] && attrs.expando == true
}
```

我们再换一个角度来看，如果是固有属性，以el[xxx] = yyy的形式赋值，再用el.getAttribute()来取值，肯定能取到东西，但自定义属性就不一样。

```
var a = document.getElementById("test");
a.title = 222
console.log(a.getAttribute("title"))        //"222"
console.log(typeof a.getAttribute("title")) //"string"

a.setAttribute("custom", "custom")
console.log(a.custom)          //undefined
console.log(typeof a.custom)   //"undefined"
```

不过要注意IE6、IE7下的特例：

```
a.setAttribute("innerHTML","xxx")
console.log(a.innerHTML)           //"xxx"
```

即使如此，我们也可以轻松绕过这个陷阱，建一个干净的同类型元素作为测试样本就行了。

```
var a = document.createElement("div")
console.log(a.getAttribute("title"))
console.log(a.getAttribute("innerHTML"))
console.log(a.getAttribute("xxx"))
console.log(a.title)
console.log(a.innerHTML)
console.log(a.xxx)
```

IE6、IE7下返回"" ,"" 、null、"" ,"" 、undefined。IE8、IE9、Chrome23、FF15和Opera12下返回null、null、null、"" ,"" 、undefined。

因此我们可以推导出这样一个方法，回答我们这一节的标题。

```
function isAttribute(attr, host){
    //有些属性是特殊元素才有的，需要用到第二个参数
    host = host || document.createElement("div");
    return host.getAttribute(attr) === null && host[attr] === v
oid 0
}
```

9.3　如何判定浏览器是否区分固有属性与自定义属性

经过社区的努力，现在大家都知道IE6、IE7不区分固有属性与自定义属性。这带来的结果是，对某个固有属性进行setAttribute，我们不需要名字映射就能生效。但如果想通过浏览器嗅探法来识别IE6、IE7，是远远不够的，因为用户可能使用旧版的标准浏览器上网。另外，我们也不得不考虑国内可恶的加壳IE浏览器。因此我们最好是通过特征侦测来判定浏览器是否支持此特性。

mootools与jQuery各自使用了两种截然不同的方法来判定。mootools以属性法设置一个自定义属性，然后通getAttribute去取，看是不是等于预设值，是就证明它不区分固有属性与自定义属性。jQuery则是先用setAttribute去设置className，然后看它是当作固有属性还是自定义属性，如果是自定

义属性，用el.className是取不到值的，具体如表9-2所示。

```
var el = document.createElement("div")
el.random = 'attribute';              //mootools
console.log(el.getAttribute("random") != 'attribute')

el.setAttribute("className", "t");     //jQuery
console.log(el.className !== "t")
```

表**9-2**

	IE6	**IE7**	**IE8**	**FF9**	**FF15**	**Chrome23**
Mootools	false	false	false	true	true	true
jQuery	false	false	true	true	true	true

要看哪个更精确，只需要IE8浏览器就行了。IE8大肆重写内核，是为了区分固有属性与自定义属性的，因为el.className应该返回undefined，导致结果为true，因此jQuery获胜。jQuery把这个特性称之为getSetAttribute，意即get/SetAttribute没有

bug；mass Framework称之为attrInnateName，意即不需要名字映射用原名就可以取值。它们都位于supports模块中。

9.4　IE的属性系统的3次演变

微软在IE4（1997年）添加setAttribute、getAttribute API。当时，DOM标准（1998年）还没有出来呢！而它的对手NS6到2000年才难产出来。

早期的DOM API于微软来说，只是它已有的一些方法的再包装，这些包装方法无法与它原来的那一套相媲美。在标准浏览器，我们是通过`document.getElementById("xxx")`来取元素节点，而在IE下，这些带ID的元素节点自动就映射成一个个全局变量，直接xxx就拿到元素节点了，很便捷，或者使用`document.all[ID]`来取，无论哪种都比标准的短；又如getElementsByTagName，IE下有`document.all.tags()`方法，此方法直到IE9还有效。而`setAttribute("xxx","yyy")`与`var ret = getAttribute("xxx")`只不过是`el.xxx = "yyy"`

与var ret = el.xxx 的另一种操作形式罢了。明白这一点我们就立即理解IE下这两个API的一些奇怪行为了。

el.setAttribute("className","aaa") 是可行的，但el.setAttribute("class", "aaa") 失败，因为我们可以用className修改类名，但不能用class。

el.setAttribute("innerHTML","<p>tes 是可行的，因为我们可以用innerHTML添加内容。

element.setAttribute("style", "background-color: #fff; color: #000;") 失败，因为style在IE下是个对象，"background-color: #fff; color: #000;" 只能作为它的cssText属性的值。

DOM level 1隔年就制定出来了。

setAttribute/getAttribute并没有微软想象得那么简单，它早期规定getAttribute必须也返回字符串，就算不存在也是空字符串。到后来，setAttribute/getAttribute会对属性名进行小写化处理。用getAttribute去取没有显式设置的固有属性时，返回默认值（多数时候它为null或空字符串）；对于没有显式设置的自定义属性，则返回undefined。于是微软傻了眼，第一次改动就是匆匆忙忙支持小写化处理，并在getAttribute方法添加第二参数，以实现DOM1的效果。

getAttribute的第二个参数有4个预设值：0是默认，照顾IE早期的行为；1属性名区分大小写；2取出源代码中的原字符串值（注，IE5～IE7对动态创建的节点没效，IE5～IE8对布尔属性无效）；4用于href属性，取得完整路径。

第二次演变是区分固有属性与自定义属性，取

类名再也不用className了。布尔属性则遵循一个奇怪的规则，只要是显式设置了就返回与属性名同名的字符串，没有则返回空字符串。笔者相信早期的标准浏览器也是这样做的。但标准浏览器很快就变脸了，统一返回用户的预设值。IE8变得两边都不讨好，尽管它一心想与标准保持一致，标榜自己才是最标准的。比如，它在当时还推出了Object.defineProperty、querySelector、postMessage等具有革命意义的新API，但它们都与W3C争吵完的结果有出入。

第三次演变，不再对属性值进行干预，用户设什么就返回什么，忠于用户的决定。对于这尘埃落定的方案，IE9终于与标准吻合了。这也说明IE这种慢吞吞的大版本发布方式已经落伍了，虽然大家都对FF的版本有微词，但人家却保住了与Chrome“叫板”的地位。

综观IE的属性系统的“悲剧”，都因为微软总想抢占先机，而又与标准同步太慢所致。

9.5　className的操作

我们操作一个属性通常只有3个选择：设置、读取、删除。但className有点特殊，它的值是可以用空格隔开，分为多个类名。因此对类名的操作变成读取、添加、删减。在“上代王者”Prototype.js，就已经把人们想要的类名操作总结出来。

```
//Prototype 1.7
  classNames: function(element) {
    return new Element.ClassNames(element);
  },

  hasClassName: function(element, className) {
    if (!(element = $(element))) return;
    var elementClassName = element.className;
    return (elementClassName.length > 0 && (elementClassName ==
 className ||
      new RegExp("(^|\\s)" + className + "(\\s|$)").test(elemen
tClassName)));
  },

  addClassName: function(element, className) {
    if (!(element = $(element))) return;
    if (!Element.hasClassName(element, className))
      element.className += (element.className ? ' ' : '') + cla
ssName;
    return element;
  },
```

```
  removeClassName: function(element, className) {
    if (!(element = $(element))) return;
    element.className = element.className.replace(
      new RegExp("(^|\\s+)" + className + "(\\s+|$)"), ' ').str
ip();
    return element;
  },
toggleClassName: function(element, className) {
    if (!(element = $(element))) return;
    return Element[Element.hasClassName(element, className) ?
      'removeClassName' : 'addClassName'](element, className);
  },
```

除了这些外，还提供了一个Element.ClassNames类，用于直接操作类名。这不禁让笔者想起HTML5提供的classList。Prototype.js基本定格了所有类名的操作方式，比如toggleClassName，这种切换方法，纷纷被其他框架抄去。如果我们把上述方法名的Name去掉，就是jQuery的那一套方法名，虽然YUI、EXT、dojo都是这么叫，如表9-3所示。

表**9-3**

框架	方法

MochiKit	hasElementClass addElementClass removeElementClass toggleElementClass setElementClass swapElementClass
Ten	hasClassName addClassName removeClassName
jQuery、avalon	hasClass addClass removeClass toggleClass
YUI3、mass、kissy	hasClass addClass removeClass toggleClass replaceClass
dojo1.8	containsClass addClass removeClass toggleClass replaceClass
EXT4	containsClass addClass removeClass toggleClass replaceClass
RightJS	hasClass addClass removeClass toggleClass radioClass setClass getClass

由此可见，这套API在业内已被认可，如果我们写框架时，命名最好不要与它们出入太大。我们亦可以把Prototype.js的实现精简一下，变成一些工具函数，在不引入类库时使用。

```
var getClass = function(ele) {
    return ele.className.replace(/\s+/," ").split(" ");
};

var hasClass =function(ele,cls){
    return -1 < (" "+ele.className+" ").indexOf(" "+cls+" ");
}
```

```
var addClass = function(ele,cls) {
    if (!this.hasClass(ele,cls))
        ele.className += " "+cls;
}

var removeClass = function(ele,cls) {
    if (hasClass(ele,cls)) {
        var reg = new RegExp('(\\s|^)'+cls+'(\\s|$)');
        ele.className=ele.className.replace(reg," ");
    }
}

var clearClass = function(ele,cls) {
    ele.className = ""
}
```

最后我们看一下avalon是如何实现这些方法的。avalon首先实现了一个伪ClassList对象，方便我们在框架里面操作元素的classList。classList是HTML5提供的最酷特性，它拥有add、remove、contains（相当于hasClass）、toggle等方法，因此我们完全可以用它来实现上面那套方法。

```
var rnowhite = /\S+/g
var fakeClassListMethods = {
    _toString: function () {
        var node = this.node
        var cls = node.className
        var str = typeof cls === 'string' ? cls : cls.baseVal
        var match = str.match(rnowhite)
        return match ? match.join(' ') : ''
```

```
    },
    _contains: function (cls) {
        return (' ' + this + ' ').indexOf(' ' + cls + ' ') > -1
    },
    _add: function (cls) {
        if (!this.contains(cls)) {
            this._set(this + ' ' + cls)
        }
    },
    _remove: function (cls) {
        this._set((' ' + this + ' ').replace(' ' + cls + ' ', '
 '))
    },
    __set: function (cls) {
        cls = cls.trim()
        var node = this.node
        if (typeof node.className === 'object') {
            //SVG元素的className是一个对象 SVGAnimatedString { bas
eVal='', animVal=''},
            //只能通过set/getAttribute操作
           node.setAttribute('class', cls)
        } else {
           node.className = cls
        }
    } //toggle存在版本差异，因此不使用它
}

function fakeClassList(node) {
    if (!('classList' in node)) {//IE10才支持classList
        node.classList = {
            node: node
        }
        for (var k in fakeClassListMethods) {
            node.classList[k.slice(1)] = fakeClassListMethods[k
]
        }
    }
    return node.classList
}
```

有了这个对象，我们实现addClass系列就是切

水果那么简单了。

```
'add,remove'.replace(avalon.rword, function (method) {
    avalon.fn[method + 'Class'] = function (cls) {
        var el = this[0] || {}
        //https://developer.mozilla.org/zh-CN/docs/Mozilla/Fire
fox/Releases/26
        if (cls && typeof cls === 'string' && el.nodeType === 1
) {
            cls.replace(rnowhite, function (c) {
                fakeClassList(el)[method](c)
            })
        }
        return this
    }
})

avalon.fn.mix({
    hasClass: function (cls) {
        var el = this[0] || {}
        return el.nodeType === 1 && fakeClassList(el).contains(
cls)
    },
    toggleClass: function (value, stateVal) {
        var isBool = typeof stateVal === 'boolean'
        var me = this
        String(value).replace(rnowhite, function (c) {
            var state = isBool ? stateVal : !me.hasClass(c)
            me[state ? 'addClass' : 'removeClass'](c)
        })
        return this
    }
})
```

有了以上这4个方法，就能应付所有关于类名的需求。

9.6 Prototype.js的属性系统

无论哪个框架类库，早年都是将它们混在一起操作。杰出的代表是Prototype.js的readAttribute、writeAttribute与jQuery的attr。我们先来看Prototype.js的伟大遗产吧。

（1）名字映射的发明。

（2）href,src的IE处理。

（3）getAttributeNode的发掘。

（4）事件钩子的处理。

（5）布尔属性的处理。

（6）style属性的IE处理。

```
// Prototype.js 1.61
readAttribute = function(element, name) {
    element = $(element);
```

```
    if (Prototype.Browser.IE) {
        var t = Element._attributeTranslations.read;
        if (t.values[name])
            return t.values[name](element, name);
        if (t.names[name])//如果要进行名字映射
            name = t.names[name];
        if (name.include(':')) {//如果XML属性
            return (!element.attributes || !element.attributes[
name]) ? null :
                    element.attributes[name].value;
        }
    }
    return element.getAttribute(name);
}
```

Prototype.js认为总是IE在拖后腿，只对IE进行调教就行了。处理方式与IE的属性系统演变史一样，对属性名的名字映射、属性值进行忠实用户的字符串还原。于是搞出Element._attributeTranslations对象，它有两大块，一是read对象用于readAttribute，二是write对象用于writeAttribute，然后每部分都有names映射列表与values函数集。

```
Element._attributeTranslations = (function() {
    //判定浏览器是否支持用class代替className
    var classProp = 'className',
            forProp = 'for',
            el = document.createElement('div');
```

```
    el.setAttribute(classProp, 'x');

    if (el.className !== 'x') {
        el.setAttribute('class', 'x');
        if (el.className === 'x') {
            classProp = 'class';
        }
    }
    el = null;
    //判定浏览器是否支持用for代替htmlFor
    el = document.createElement('label');
    el.setAttribute(forProp, 'x');
    if (el.htmlFor !== 'x') {
        el.setAttribute('htmlFor', 'x');
        if (el.htmlFor === 'x') {
            forProp = 'htmlFor';
        }
    }
    el = null;

    return {
        read: {
            names: { //名字映射
                'class': classProp,
                'className': classProp,
                'for': forProp,
                'htmlFor': forProp
            },
            values: {//钩子函数对象
                //处理普通自定义属性
                _getAttr: function(element, attribute) {
                    return element.getAttribute(attribute);
                },
                //处理src,href等路径相关的属性
                _getAttr2: function(element, attribute) {
                    return element.getAttribute(attribute, 2);
                },
                //处理ID相关的固有属性
                _getAttrNode: function(element, attribute) {
                    var node = element.getAttributeNode(attribu
te);
                    return node ? node.value : "";
                },
                //处理事件 onXXX
                _getEv: (function() {
```

```
                //...略
                    return f;
                })(),
                //处理readOnly, disable等布尔属性
                _flag: function(element, attribute) {
                    return $(element).hasAttribute(attribute) ?

                    attribute : null;
                },
                //处理样式
                style: function(element) {
                    return element.style.cssText.toLowerCase();
                },
                //处理title
                title: function(element) {
                    return element.title;
                }
           }
        }
    }
})();
```

上面是读方法的处理，接着是写方法。

```
Element._attributeTranslations.write = {
    names: Object.extend({
        cellpadding: 'cellPadding',
        cellspacing: 'cellSpacing'
    }, Element._attributeTranslations.read.names),
    values: {
        checked: function(element, value) {
            element.checked = !!value;
        },
        style: function(element, value) {
            element.style.cssText = value ? value : '';
        }
    }
};

Element._attributeTranslations.has = {};
```

```
// 处理不规则的属性名转换
$w('colSpan rowSpan vAlign dateTime accessKey tabIndex ' +
        'encType maxLength readOnly longDesc frameBorder').each
(function(attr) {
    Element._attributeTranslations.write.names[attr.toLowerCase
()] = attr;
    Element._attributeTranslations.has[attr.toLowerCase()] = at
tr;
});
```

从读方法的代码可看到，对于for、class的映射，它是使用特性侦测来实现的。然后对不同的属性，采取不同的方法取值，其实就是一个适配器。Element._attributeTranslations.values就是个bug发掘器，发现绝大部分的bug，以后jQuery的工作就是进行改良与深挖。

IE6～IE8表示URL的属性会有一些返回补全的改过编码的路径，如action、background、BaseHref、cite、codeBase、data、dynsrc、href、longDesc、lowsrc、pluginspage、profile、src、url与vrml。但一般框架只处理href、src，处理方式很简单，即将getAttriubte的第二个参数设为2。

```
<a href="index.html">home</a>
<script>
var link = document.getElementsByTagName('a')[0];
link.getAttribute('href') // "http://www.cnblogs.com/rubylouvre
/index.html";
link.getAttribute('href',2) //"index.html";
</script>
```

至于编码的情况就更复杂了，我们可以使用`<a href="${链接1}"` 作为实验样本，如表 9-4所示。

表**9-4**

	href	**getAttribute("href")**	**getAttribute("href", 2)**
IE6	转绝对，不编码	转绝对，不编码	正常
IE7	转绝对，汉字不编码，特殊符号编码	转绝对，汉字不编码，特殊符号编码	正常
IE8	转绝对，汉字不编码，特殊符号编码	正常	正常
Firefox 3.0+	转绝对，全部编码	正常	正常
Chrome 2.0	转绝对，全部编码	正常	正常

Safari 4.0	转绝对，汉字编码，特殊符号不编码	正常	正常
Opera 10.0	转绝对，汉字编码，特殊符号不编码	正常	正常

其中，“正常”的意思是，得到href属性里原始链接，不自动转绝对地址，汉字和符号都不编码。

在IE6、IE7中，form元素用getAttribute取属性值，可能得到它辖下的ID值或name值相同的表单元素。在Prototype.js下，只处理比较常见的action属性。下面的例子，在IE6、IE7下都是返回元素节点。

```
<form  action="#" >
        <input id="name" >
        <input id="action" >
        <input name="id"  >
        <input name="length"  >
        <input id="xxx" >
        <input id="yyy" >
 </form>

var el = document.getElementsByTagName("form")[0]
alert(el.getAttribute("action"))
alert(el.getAttribute("id"))
alert(el.getAttribute("name"))
```

```
alert(el.getAttribute("length"))
alert(el.getAttribute("xxx"))
alert(el.getAttribute("yyy"))
```

Prototype.js于是发掘到getAttributeNode方法。当然也可以用attriubtes[xxx]方法，只要得到特性节点就好办了。

对于事件钩子，Prototype.js可谓是费力来修改回调的toString。不过在此之前，它还判定元素是否支持事件。如果Prototype.js的作者再多做些测试，其实getAttributeNode与attriubtes[xxx]方式也能取到正确值。

对于布尔属性，Prototype.js是用_flag内部方法去取的。如果el[xxx]是返回true的情况，直接返回与属性同名的字符串，否则返回字符串。这是一个没有办法的办法，因为对于布尔属性，IE6～IE8都无法取得用户预设值，无论是getAttribute的第二个参数方式，还是getAttributeNode方法、

attributes[xxx]方式以及outerHTML的字符串裁剪方式。后来标准浏览器改变游戏规则了，对于真值的布尔属性不再返回同名字符串，假值不再返回空字符串，忠于用户输入。于是，Prototype.js的readAttribute方法在不同浏览器中的返回值就不同了。jQuery做出的处理是，编写了一个正则，用于匹配布尔属性，统一所有浏览器在真值的情况下返回同名字符串，假值返回undefined。

```
rboolean = /^(?:autofocus|autoplay|async|checked|controls|defer
|disabled|hidden|loop |multiple|open|readonly|required|scoped|s
elected)$/i
```

但谁也无法保证，这正则已经囊括所有布尔属性，而且大多数布尔只对个别元素有效，因此这是晕招。不过比起Prototype.js已是很大进步，后者只处理表单元素的几个特定属性：disabled、checked、readonly、multiple。

IE处理style属性，即统一转换它的cssText属

性，这个所有框架都一样。

在写入属性值时，Prototype.js察觉到属性有两种形态。一种是存在于HTML标签内的，不区分大小写，不过现在浏览器都统一将它们小写化。对于存在多个字母个数与排列顺序相同但大小写不同的元素，标准浏览器会做合并处理，只保留第一个属性。DOM形态是区分大小写，并且对保留字进行回避，如著名的class、for，有的是两个单词组成，需要转换成驼峰风格，有的则完全是没有规则。正如英语单词的单复数转换，一开始没有规范，大家都照抄网景的，然后又搞些独创的属性，经过这么多的发展，我们一个小小映射能应对90%的特殊情况就是谢天谢地了。

Prototype.js列出了15个需要映射的情况，而jQuery的只有12个，其中encoding这映射在标准浏览器中还不会出现。这一数字在mass Framework飙

升了36个，算是目前兼容较全面的。

Prototype.js干这活的对象为Element._attributeTranslations.write.names，一共处理以下属性：accessKey、cellPadding、cellSpacing、className、colSpan、dateTime、encType、frameBorder、htmlFor、longDesc、maxLength、readOnly、rowSpan、tabIndex、vAlign。

jQuery干这活的对象为$.propFix，一共处理以下属性：cellPadding、cellSpacing、className、colSpan、colSpan、contentEditable、encoding、frameBorder、htmlFor、rowSpan、rowSpan、useMap。

mass Framework干这活的对象为$.propMap，一共处理以下属性：acceptCharset、accessKey、allowTransparency、bgColor、cellPadding、

cellSpacing、ch、chOff、className、codeBase、codeType、colSpan、contentEditable、dateTime、defaultChecked、defaultSelected、defaultValue、encoding、frameBorder、htmlFor、httpEquiv、isMap、longDesc、marginHeight、marginWidth、maxLength、noHref、noResize、noShade、readOnly、rowSpan、tabIndex、useMap、vAlign、vSpace、valueType。

Prototype.js的writeAttribute方法是在Prototype.js 1.6中出现的，决定了传入null与false属性值进行删除操作的游戏规则，但是被jQuery 1.6抄去了。然而事实上，传入false消去当前布尔属性的效果，也只有removeAttribute一途，这是聪明的做法。但至于把它应用到固有属性，就需慎重考虑了。反过来想，正因为Prototype.js想用一个API同时处理固有属性与自定义属性，才导致删除操作也不得不保持一致。jQuery追随Prototype.js的步伐，

于是掉进同一个陷阱。

```
//Prototype.js1.61
writeAttribute = function(element, name, value) {
    element = $(element);
    var attributes = {}, t = Element._attributeTranslations.wri
te;
    if (typeof name == 'object')
        attributes = name;
    else
        attributes[name] = Object.isUndefined(value) ? true : v
alue;
    for (var attr in attributes) {
        name = t.names[attr] || attr;
        value = attributes[attr];
        if (t.values[attr])
            name = t.values[attr](element, value);
        if (value === false || value === null)
            element.removeAttribute(name);
        else if (value === true)
            element.setAttribute(name, name);
        else
            element.setAttribute(name, value);
    }
    return element;
}
```

9.7 jQuery的属性系统

早期的jQuery针对当时如日中天的Prototype.js提出了一个重要的口号：它更小。笔者还是希望能带来一点启迪。jQuery的属性系统也是经年累月，量变引发质变的结果。

```
// jQuery1.01
attr: function(elem, name, value){
        var fix = {
            "for": "htmlFor",
            "class": "className",
            "float": "cssFloat",
            innerHTML: "innerHTML",
            className: "className",
            value: "value",
            disabled: "disabled"
        };

        if ( fix[name] ) {
            if ( value != undefined ) elem[fix[name]] = value;
            return elem[fix[name]];
        } else if ( elem.getAttribute ) {
           if ( value != undefined ) elem.setAttribute( name, v
alue );
            return elem.getAttribute( name, 2 );
        } else {
            name = name.replace(/-([a-z])/ig,function(z,b){retu
rn b.toUpperCase();});
            if ( value != undefined ) elem[name] = value;
            return elem[name];
        }
},
```

这个方法然后不断膨胀，加入从Prototype.js发掘出来的特殊属性处理，以及社区提交的补丁。到jQuery 1.5.2，这个attr方法已经接近一百行的规模。于是jQuery 1.6模块，在jQuery 1.5中的CSS模块想出了好方法后，把cssHooks适配器机制移值过来，解决了扩展的难题。

在jQuery 1.6中存在4个适配器，从单词直译过来，就是钩子：formHooks、attrHooks、propHooks、valHooks。formHook是在attr方法中对付旧版本IE的form元素用的。到jQuery 1.6.1，增加一个boolHooks对付布尔属性。到jQuery 1.6.3，人们发现IE大多数情况使用getAttriubteNode就能取到正确值，因此对formHooks重构一下，更名为nodeHooks，便形成今天jQuery的属性系统，如图9-1所示。

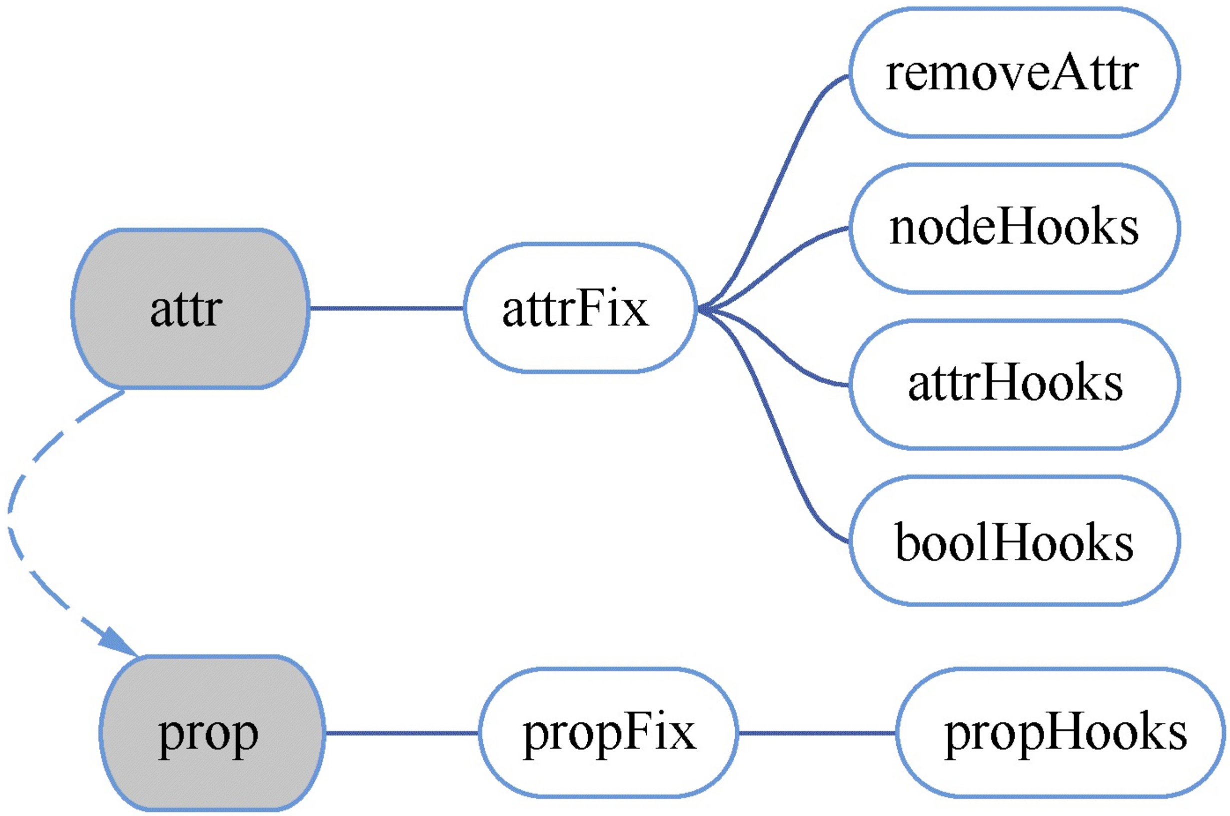


图9-1

新生的prop方法异常简单，复杂性都移到钩子。古老的attr方法则无比复杂，兼任读、写、删除3职。由于IE的情况不得不动用到3个钩子，钩子只处理有问题的属性，不处理一般情况。

```
//jquery1.83
prop: function( elem, name, value ) {
    var ret, hooks, notxml,  nType = elem.nodeType;
```

```
    if ( !elem || nType === 3 || nType === 8 || nType === 2 ) {
        return;//跳过注释节点、文本节点、特性节点
    }

    notxml = nType !== 1 || !jQuery.isXMLDoc( elem );
    if ( notxml ) {//如果是HTML文档的元素节点
        name = jQuery.propFix[ name ] || name;
        hooks = jQuery.propHooks[ name ];
    }

    if ( value !== undefined ) {//写方法
        if ( hooks && "set" in hooks && (ret = hooks.set( elem,
 value, name )) !== undefined ) {
            return ret;//处理特殊情况

        } else {//处理通用情况
            return ( elem[ name ] = value );
        }

    } else {//读方法
        if ( hooks && "get" in hooks && (ret = hooks.get( elem,

        name )) !== null ) {
            return ret;//处理特殊情况
        } else {//处理通用情况
            return elem[ name ];
        }
    }
},
attr: function( elem, name, value ) {
    var ret, hooks, notxml, nType = elem.nodeType;

    if ( !elem || nType === 3 || nType === 8 || nType === 2 ) {
        return;//跳过注释节点、文本节点、特性节点
    }
    if ( typeof elem.getAttribute === "undefined" ) {
        return jQuery.prop( elem, name, value );//文档与window,
只能使用prop
    }

    notxml = nType !== 1 || !jQuery.isXMLDoc( elem );
    if ( notxml ) {//如果是HTML文档的元素节点
        name = name.toLowerCase();//决定用哪一个钩子
        hooks = jQuery.attrHooks[ name ] || ( rboolean.test( na
me ) ? boolHook : nodeHook );
```

```
    }

    if ( value !== undefined ) {
        if ( value === null ) {//模仿Prototype.js，移除属性
            jQuery.removeAttr( elem, name );

        } else if ( hooks && notxml && "set" in hooks &&
            (ret = hooks.set( elem, value, name )) !== undefine
d ) {
            return ret;//处理特殊情况

        } else {//处理通用情况
            elem.setAttribute( name, value + "" );
            return value;
        }
    } else if ( hooks && notxml && "get" in hooks && (ret = hoo
ks.get( elem, name )) !== null ) {
        return ret;//处理特殊情况

    } else {//处理通用情况
        ret = elem.getAttribute( name );
        return ret === null ? undefined :  ret;
    }
},
```

每个钩子的结构也不一样，propHooks、attrHooks里面是以属性命名的对象，里面可能存在get方法、set方法或两者都有。boolHooks、nodeHooks则只有set、get两个方法，如图9-2所示。

钩子在所有特殊情况下都命中的结构图。目前在Firefox下命中最少，在IE6、IE7命中最多。

jQuery对属性系统的主要贡献是发明更多兼容性问题与解决方法，具体如下。

（1）tabindex的取值问题。tabindex默认情况下只对表单元素与链接有效，对于这些元素没有显式设置会返回0，对于像DIV这样普通元素返回-1，但IE都是返回0。jQuery对此做了统一。

（2）Safari下，option元素的selected取值问题，必须向上访问一下select元素才得到结果。

（3）表单元素的value属性的操作。由于表单元素种类够多，存在严重的兼容性问题，jQuery为此下了很大工夫让我们write less，do more！

但jQuery的属性系统也存在明显的缺憾，具体如下。

（1）名字映射是一种穷举机制，但jQuery也有待完善，让attrFix、propFix都没有尽责。

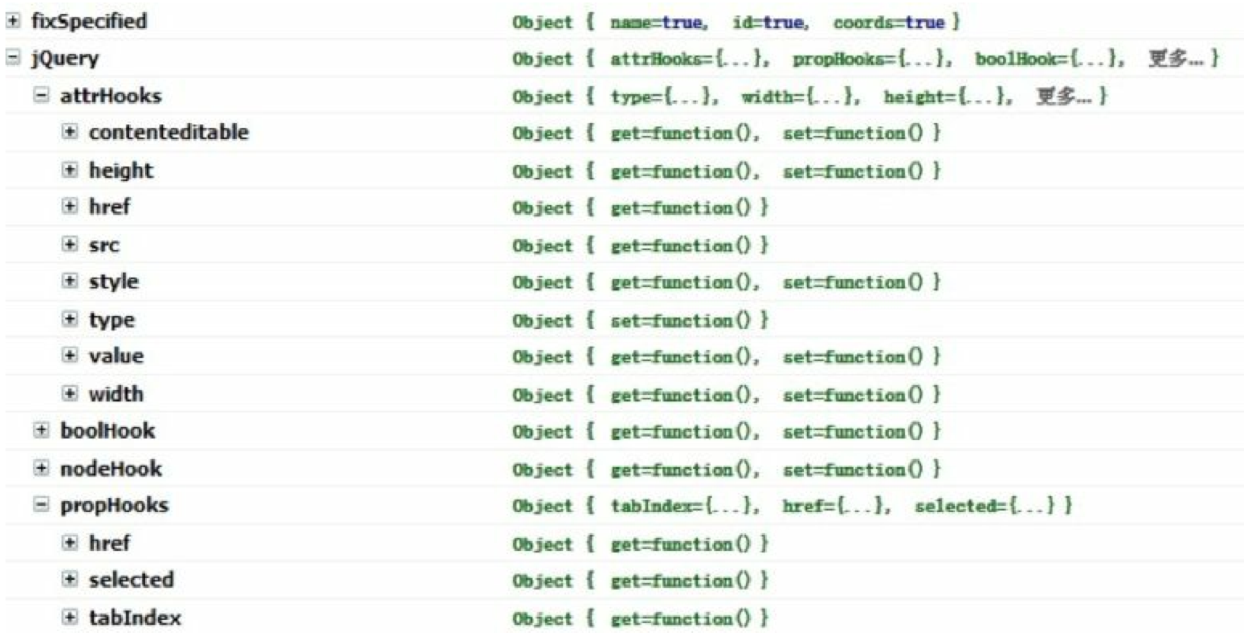


图9-2

（2）对布尔属性的判定存在硬编码，准确率极低。

（3）它添加了一个与removeAttr对称的removeProp方法，但里面的实现用到delete操作符，在chrome真的是把固有属性从原型中删掉，导致下次赋值时出错。

（4）nodeHooks是使用getAttributeNode实现的，虽然能应对所有自定义属性，但判定某些固有

属性是否为显式属性时，却不得不使用fixSpecified补漏洞。可惜fixSpecified对象作为一种穷举机制，在吝惜类库体积的jQuery中注定水土不服。

（5）因为旧版本IE7不能修改表单元素的type属性，就阻止我们在所有浏览器修改type，这种行为很难服众。比如密码域的replaceholder模拟、苹果应用上那些人性化的掩码，就需要我们修改type。

接着下来重点介绍avalon的实现，它在jQuery的属性系统上做了大量改进，克服了以上缺点。

9.8 avalon的属性系统

avalon的属性系统其实是供其ms-attr指令内部使用的，能同时处理HTML元素节点、VML元素节点及SVG元素节点。其中VML读写属性时使用经典的'el.xxx = yyy'来处理，而SVG则必须使用set/getAttribute方法。

在avalon内部有3个模块来实现属性系统。首先是propMap模块，它提供一个对象，用于名字映射，如图9-3所示。

```
https://github.com/RubyLouvre/avalon/tree/2.1.6/src/dom/attr
var propMap = {//不规则的属性名映射
    'accept-charset': 'acceptCharset',
    'char': 'ch',
    charoff: 'chOff',
    'class': 'className',
    'for': 'htmlFor',
    'http-equiv': 'httpEquiv'
}
//所有已知的布尔属性
var bools = ['autofocus,autoplay,async,allowTransparency,checke
d,controls',
    'declare,disabled,defer,defaultChecked,defaultSelected,',
    'isMap,loop,multiple,noHref,noResize,noShade',
    'open,readOnly,selected'
```

```
].join(',')

bools.replace(/\w+/g, function (name) {
    propMap[name.toLowerCase()] = name
})

//驼峰名
var anomaly = ['accessKey,bgColor,cellPadding,cellSpacing,codeB
ase,codeType,colSpan',
    'dateTime,defaultValue,contentEditable,frameBorder,longDesc
,maxLength'+
    'marginWidth,marginHeight,rowSpan,tabIndex,useMap,vSpace,va
lueType,vAlign'
].join(',')
anomaly.replace(/\w+/g, function (name) {
    propMap[name.toLowerCase()] = name
})

module.exports = propMap
```

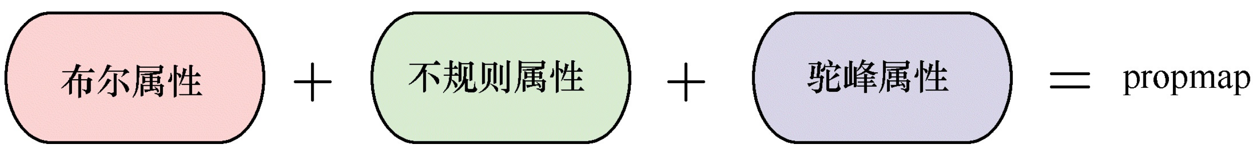


图9-3

然后是判定元素是否VML，之前也给出过。

```
function isVML(src) {
    var nodeName = src.nodeName
    return nodeName.toLowerCase() === nodeName && src.scopeName
 && src.outerText === ''
}
module.exports = isVML
```

最后是attrUpdate模块，里面只有一个函数。

```
var propMap = require('./propMap')
var isVML = require('../html/isVML')
var rsvg =/^\[object SVG\w*Element\]$/
var ramp = /&amp;/g

function attrUpdate(node, vnode) {
    if (!node || node.nodeType !== 1 ) {
        return
    }
    vnode.dynamic['ms-attr'] = 1
    var attrs = vnode['ms-attr']
    for (var attrName in attrs) {
        var val = attrs[attrName]
        // 处理路径属性
        /* istanbul ignore if*/
        if (attrName === 'href' || attrName === 'src') {
            if (!node.hasAttribute) {
                val = String(val).replace(ramp, '&')
                //处理IE67自动转义的问题
            }
            node[attrName] = val
            /* istanbul ignore if*/
            if (window.chrome && node.tagName === 'EMBED') {
                var parent = node.parentNode
                          //#525  chrome1-37下embed标签动态设置src
不能发生请求
                var comment = document.createComment('ms-src')
                parent.replaceChild(comment, node)
                parent.replaceChild(node, comment)
            }
            //处理HTML5 data-*属性
        } else if (attrName.indexOf('data-') === 0) {
            node.setAttribute(attrName, val)

        } else {
            var propName = propMap[attrName] || attrName
            if (typeof node[propName] === 'boolean') {
                node[propName] = !!val

                //布尔属性必须使用el.xxx = true|false方式设值
                //如果为false, IE全系列下相当于setAttribute(xxx,'')
```

```
,
                //会影响到样式,需要进一步处理
            }

            if (val === false) {//移除属性
                node.removeAttribute(propName)
                continue
            }
            //SVG只能使用setAttribute(xxx, yyy), VML只能使用node.x
xx = yyy ,
            //HTML的固有属性必须node.xxx = yyy

            var isInnate = rsvg.test(node) ? false :
                    (!avalon.modern && isVML(node)) ? true :
                    attrName in node.cloneNode(false)
            if (isInnate) {
                node[propName] = val + ''
            } else {
                node.setAttribute(attrName, val)
            }
        }
    }
}

module.exports = attrUpdate
```

图9-4已经涉及MVVM的高阶内容了。attrUpdate为一个视图刷新函数，node为真实元素节点，vnode为虚拟节点。虚拟节点可以理解为一个普通的JS对象就行了，它用来描述某个节点的结构与内容，但比真实节点是轻量多。vnode上有一个ms-attr属性，它或许是一个对象，也或许不存在。

此对象大概如下。

```
vnode["ms-attr"]= {
   title: "aaa",
   ddd: false,
   eee: true
}
```

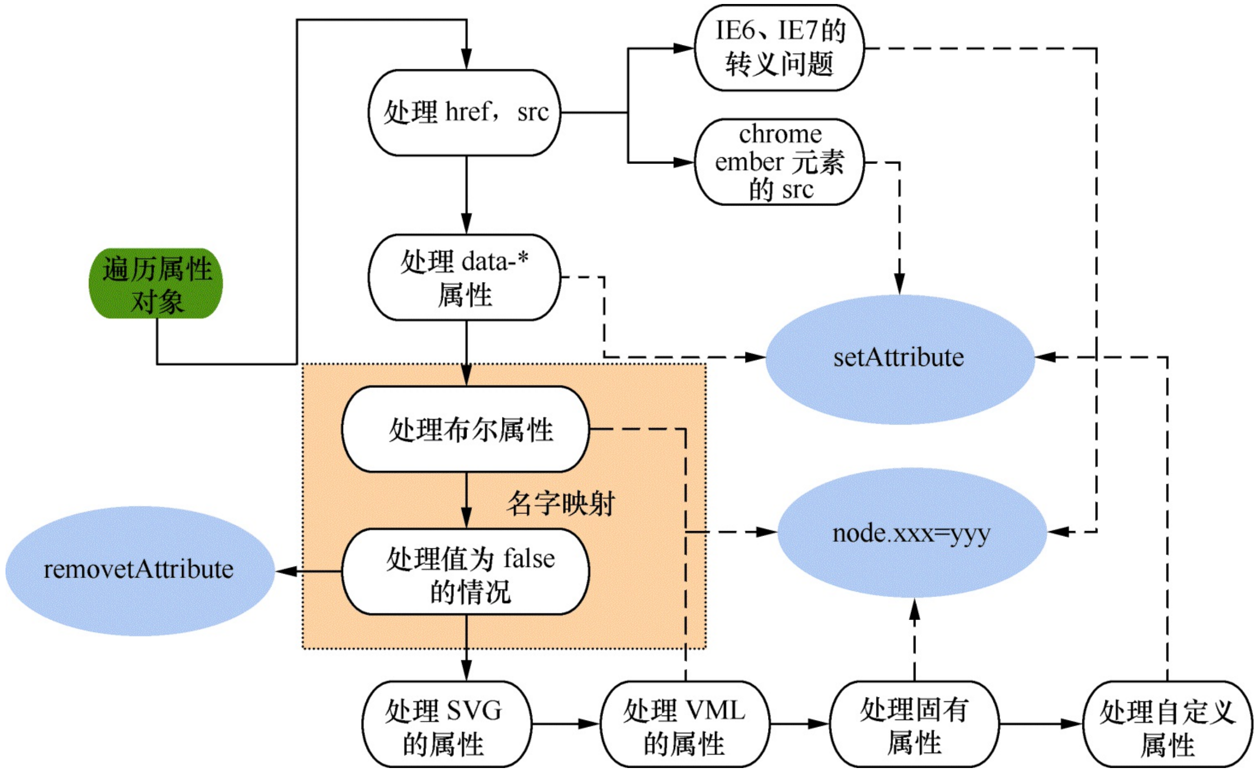


图9-4

当某一个属性的值为false时，表示我们要移除此属性，其他操作则进行写操作。avalon内部没有对普通属性的读操作，因此在MVVM中，这个不太常用。

9.9 value的操作

上一节说avalon内部没有对普通属性的读操作，但对于value就不能蒙混过关了，因为value属性是要提交到后台的。由于其重要性，于是我们单独拿出来讲解。

value属性，一般而言，只有表单元素的value才对我们有用。问题是，表单元素的种类非常多，每一个取法与赋值各有不同，因此我们最好在内部使用一个适配器来实现它。有关表单元素取value值与大部分浏览器的兼容性问题被jQuery发现并消灭了，avalon只是在字节上做压缩处理。

下面分别描述每个表单元素的情况。

select元素，它的value值就是其被选中的option孩子的value值。不过，select有两种形态，一种type

为select-one，另一种为select-multiple，就是当用户显式设置了multiple属性。在多选形态下，我们可以在Windows系统下按住Ctrl键进行多选，Mac系统下按住Command键进行多选。

option元素，它的value值可以是value属性的值，亦可以是其中间的文本，换言之是innerText。当用户没有显式设置value属性时，它就取innerText，不过这个innerText要用text属性来取，就像script标签那样。可能有人会问，为什么不用innerHTML呢？因为这个option元素的text属性比innerHTML多做了一个trim操作，去掉两边的空白（旧版本IE的innerHTML会做trim操作，标准浏览器不会）。那么如何判定它是显示设置了value属性呢？这将在10.1节中提到，元素节点有个attribute属性，它是个类数组对象，里面是一个个对象，每个对象拥有value、name、specified、ownerElement等一大堆属性，我们判定specified是否为true就行了。

IE8与其他浏览器，我们还可以使用hasAttribute方法来判定。

button元素，它的取值情况与option元素有点类似但又不尽然。在IE6、IE7中，它是取元素的innerText，到IE8时它才与其他浏览器保持一致，取value属性的值。不过在标准浏览器下，button标签只有当其为提交按钮，并且点击它时，才会提交其自身的value值。我们应该统一返回其value值。

checkbox、radio在设置value时，应该考虑对checked属性的修改。

因此为了对应这么多奇怪情况，我们又需要引入钩子对象，像上一章的cssHooks那样，弄一个valHooks。

```
avalon.fn.val = function (value) {
    var node = this[0]
    if (node && node.nodeType === 1) {
        var get = arguments.length === 0
        var access = get ? ':get' : ':set'
        var fn = valHooks[getValType(node) + access]
```

```
        if (fn) {
            var val = fn(node, value)
        } else if (get) {
            return (node.value || '').replace(/\r/g, '')
        } else {
            node.value = value
        }
    }
    return get ? val : this
}
```

里面有一个getValType方法，用来告诉我们接着下来该调用什么钩子方法。

```
function getValType(elem) {
    var ret = elem.tagName.toLowerCase()
    return ret === 'input' && /checkbox|radio/.test(elem.type)
? 'checked' : ret
}
```

接着逐个击破。

首先option元素的value值，如果用户没有写value属性，会取其innerText作value值的。

```
var roption = /^<option(?:\s+\w+(?:\s*=\s*(?:"[^"]*"|'[^']*'|[^
\s>]+))?)*\s+value[\s=]/i

valHooks['option:get'] =  avalon.msie ? function (node) {
        //在IE11及W3C，如果没有指定value，那么node.value默认为node.t
```

```
ext（存在trim作），但
        //IE9、IE10则是取innerHTML(没trim操作)
        //specified并不可靠，因此通过分析outerHTML判定用户有没有显示定
义value
        return roption.test(node.outerHTML) ? node.value : node
.text.trim()
    } : function (node) {
        return node.value
}
```

然后是select元素的value值的处理，它会同步底下的option元素。

```
valHooks['select:get'] = function (node, value) {
        var option, options = node.options,
                index = node.selectedIndex,
                getter = valHooks['option:get'],
                one = node.type === 'select-one' || index < 0,
                values = one ? null : [],
                max = one ? index + 1 : options.length,
                i = index < 0 ? max : one ? index : 0
        for (; i < max; i++) {
           option = options[i]
           //IE6-9在reset后不会改变selected，需要改用i === index判定
           //我们过滤所有disabled的option元素，但在safari5下，
           //如果设置optgroup为disable，那么其所有孩子都disable
           //因此当一个元素为disable，需要检测其是否显式设置了disable及
其父节点的disable情况
           if ((option.selected || i === index) && !option.disa
bled &&
                   (!option.parentNode.disabled || option.paren
tNode.tagName !== 'OPTGROUP')
                    ) {
                value = getter(option)
                if (one) {
                    return value
                }
                //收集所有selected值组成数组返回
```

```
                values.push(value)
            }
        }
        return values
    }

valHooks['select:set'] = function (node, values, optionSet) {
        values = [].concat(values) //强制转换为数组
        var getter = valHooks['option:get']
        for (var i = 0, el; el = node.options[i++]; ) {
            if ((el.selected = values.indexOf(getter(el)) > -1)
) {
                optionSet = true
            }
        }
        if (!optionSet) {
            node.selectedIndex = -1
        }
    }
}
```

radio与checkbox的处理如下。

```
//checkbox的value默认为on，唯有Chrome 返回空字符串
if (!support.checkOn) {
    valHooks["checked:get"] = function(node) {
        return node.getAttribute("value") === null ? "on" : nod
e.value;
    };
}
//处理单选框、复选框在设值后checked的值
valHooks["checked:set"] = function(node, name, value) {
    if (Array.isArray(value)) {
        return node.checked = !!~value.indexOf(node.value);
    }
}
```

9.10 总结

自定义属性与固有属性的区别如下。

（1）元素节点在创建时，就拥有property，而自定义属性则是后来添加的。

（2）在W3C规范中，只有在HTML标签定义的属性才会出现在元素的attributes对象，这些属性又叫显式属性。

（3）自定义属性在初始化时同步同名的固有属性。

（4）自定义属性的值都是字符串。

（5）大多数情况下，我们修改自定义属性时，会同步固有属性，但修改固有属性时，不会影响自定义属性。

第10章　PC端的事件系统


Sold
NEWS
FOR SALE
$
www.taopic.com

第10章　PC端的事件系统

事件系统是很大的一块，因为我们与PC之间的交互行为就存在各种形式，比如我们可以通过键盘进行交互，鼠标进行交互，或者直接触摸屏幕进行交互。此外，即便你什么也不做，PC也单方面告诉你现在是处于某种状态，如页面的DOM树构建完毕会抛出一个事件，所有资源加载完毕也会抛出一个事件，图片加载成功或失败也会抛出事件，CSS3动画开始或结束也会抛出事件，甚至一个元素插入DOM树或移出DOM树，它的某个属性值发生改变了，现在都有对应的事件抛出来了。所有事件的设计还是很一致的，都有一个参数，为一个事件对象，会告诉你事件源在哪里，它是什么事件。

在PC端，事件只有些许兼容性问题，浏览器已经提供好“弹药”，我们只需要稍作改造就可以用。而在移动端，所有能用的事件都需要合成出来。这

也是本书将它们将分开来说的缘故。

告诉大家一个技巧，后端遇到复杂的问题都是中间再加一层，而我们处理事件的兼容问题，则是再包一个函数。外包一个处理函数，这点类似于AOP，将重复的部分抽取出来（如事件对象的标准化处理，this对象的绑定，频繁调度事件的函数节流处理……），这个编程技巧以后我们会经常用到。

PC端常用事件一览，如图10-1所示。

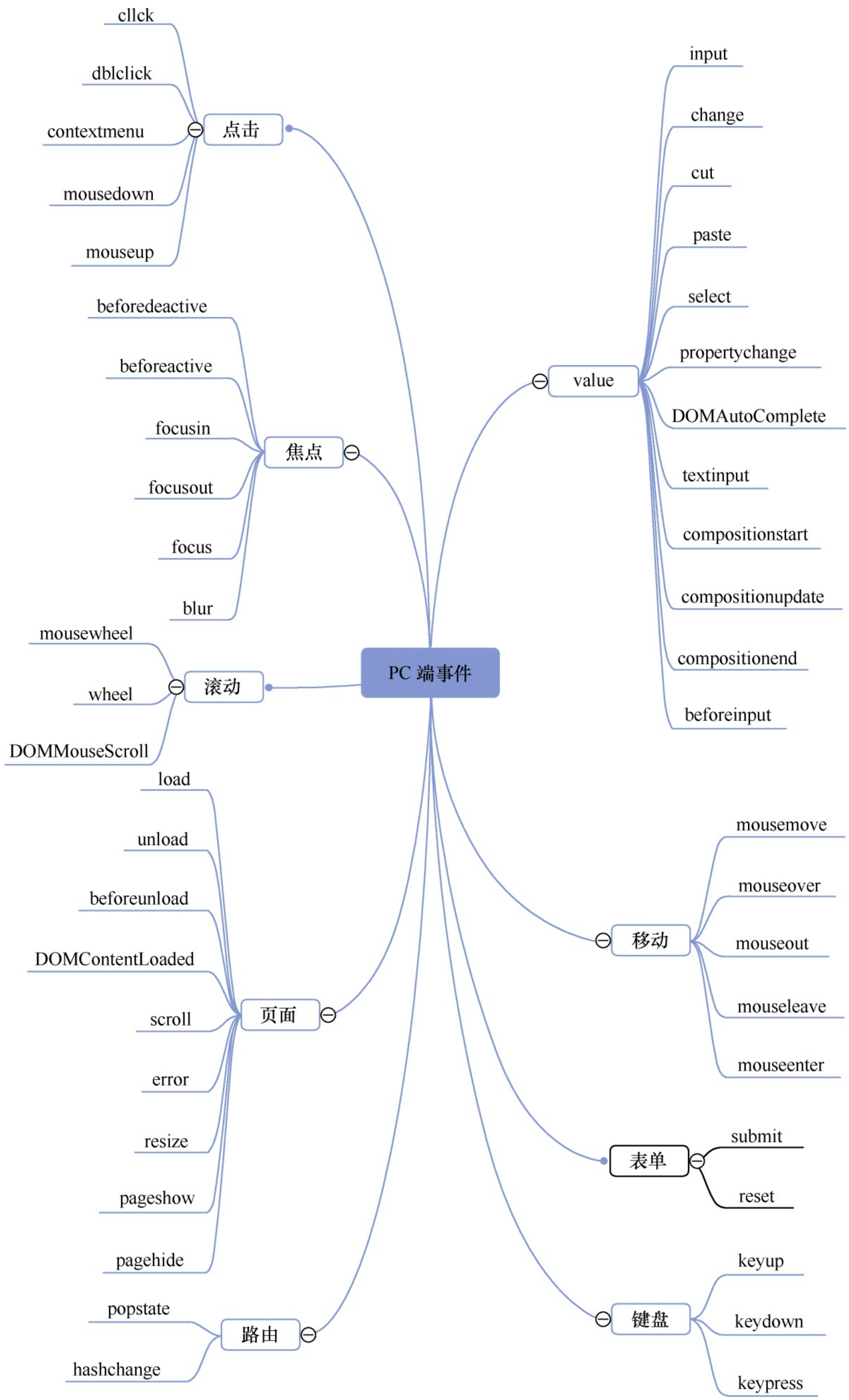

PC 端事件
点击
cllck
dblclick
contextmenu
mousedown
mouseup
焦点
beforedeactive
beforeactive
focusin
focusout
focus
blur
滚动
mousewheel
wheel
DOMMouseScroll
页面
load
unload
beforeunload
DOMContentLoaded
scroll
error
resize
pageshow
pagehide
路由
popstate
hashchange
value
input
change
cut
paste
select
propertychange
DOMAutoComplete
textinput
compositionstart
compositionupdate
compositionend
beforeinput
移动
mousemove
mouseover
mouseout
mouseleave
mouseenter
表单
submit
reset
键盘
keyup
keydown
keypress

图10-1

10.1　原生API简介

事件系统是一个框架非常重要的部分，用于响应用户的各种行为。

浏览器提供了3种层次的API。最原始的是写在元素标签内。再次是在脚本中，以`el.onXXX=function`绑定的方式，通称为DOM0事件系统。最后是多投事件系统，一个元素的同一类型事件可以绑定多个回调，通称为DOM2事件系统。由于浏览器大战，现存在两套API。

IE与Opera方法如下。

```
绑定事件：el.attachEvent("on"+type，callback)
卸载事件：el.detachEvent("on"+type，callback)
创建事件：document.createEventObject()
派发事件：el.fireEvent(type, event)
```

W3C方法如下。

```
绑定事件：el.addEventListener(type, callback, [phase])
卸载事件：el.removeEventListener(type, callback, [phase])
创建事件：el.createEvent(types)
初始化事件：event.initEvent()
派发事件：el.dispatchEvent(event)
```

从API的数量与形式来看，W3C提供的复杂很多，相对应也强大很多，下面笔者会逐一详述。

首先呈上几个简单的实现。如果是简单的页面，就用简单的方式打发，没有必要动用框架。不过，事实上，整个事件系统就建立在它们的基础上。

```
function addEvent(el, type, callback, useCapture ){
    if(el.dispatchEvent){//W3C方式优先
        el.addEventListener( type, callback, !!useCapture  );
    }else {
        el.attachEvent( "on"+type, callback );
}
    return callback;//返回callback方便卸载时用
}

function removeEvent(el, type, callback, useCapture){
    if(el.dispatchEvent){//W3C方式优先
        el.removeEventListener( type, callback, !!useCapture  )
;
    }else {
        el.detachEvent( "on"+type, callback );
    }
}
```

```
function fireEvent(el, type, args, event){
    args = args || {}
    if(el.dispatchEvent){//W3C方式优先
        event = document.createEvent("HTMLEvents");
        event.initEvent(type, true, true );
    }else {
        event = document.createEventObject();
    }
    for(var i in args)  if(args.hasOwnProperty(i)){
        event[i] = args[i]
    }
    if(el.dispatchEvent){
        el.dispatchEvent(event);
    }else{
        el.fireEvent('on'+type,event)
    }
}
```

这3个方法也经常出现于各种笔试中，因此大家要切记。

10.2 onXXX绑定方式的缺陷

onXXX既可以写在HTML标签内，也可以独立出来，作为元素节点的一个特殊属性来处理。不过作为一种古老的绑定方式，它很难预测到后来人对这方面的扩展。

总结起来有以下不足。

（1）onXXX对DOM3新增事件或FF某些私有实现无法支持，主要有以下事件：

```
DOMActivate
DOMAttrModified
DOMAttributeNameChanged
DOMCharacterDataModified
DOMContentLoaded
DOMElementNameChanged
DOMFocusIn
DOMFocusOut
DOMMouseScroll
DOMNodeInserted
DOMNodeInsertedIntoDocument
DOMNodeRemoved
DOMNodeRemovedFromDocument
DOMSubtreeModified
MozMousePixelScroll
```

不过稍安勿躁，上面的事件就算是框架，也只用到DOMContentLoaded与DOMMouseScroll。DOMContentLoaded用于检测DomReady，DOMMouseScroll用于在FF模拟其他浏览器的mousewheel事件。

（2）onXXX只允许元素每次绑定一个回调，重复绑定会冲掉之前的绑定。

（3）onXXX在IE下回调没有参数，在其他浏览器下回调的第一个参数是事件对象。

（4）onXXX只能在冒泡阶段可用。

10.3　attachEvent的缺陷

attachEvent是微软在IE5添加的API，Opera也支持。对于onXXX方式，它可以允许同一元素同种事件绑定多个回调，也就是所谓的多投事件机制。但它带来的麻烦只多不少，存在以下几点缺陷。

（1）IE下只支持微软系的事件，DOM3事件一概不能用。

（2）IE下attchEvent回调中的this不是指向被绑定元素，而是window！

（3）IE下同种事件绑定多个回调时，回调并不是按照绑定时的顺序依次触发的！

（4）IE下event事件对象与W3C的存在太多差异了，有的无法对上号，比如currentTarget。

（5）IE还是只支持冒泡阶段（如图10-2所示）。

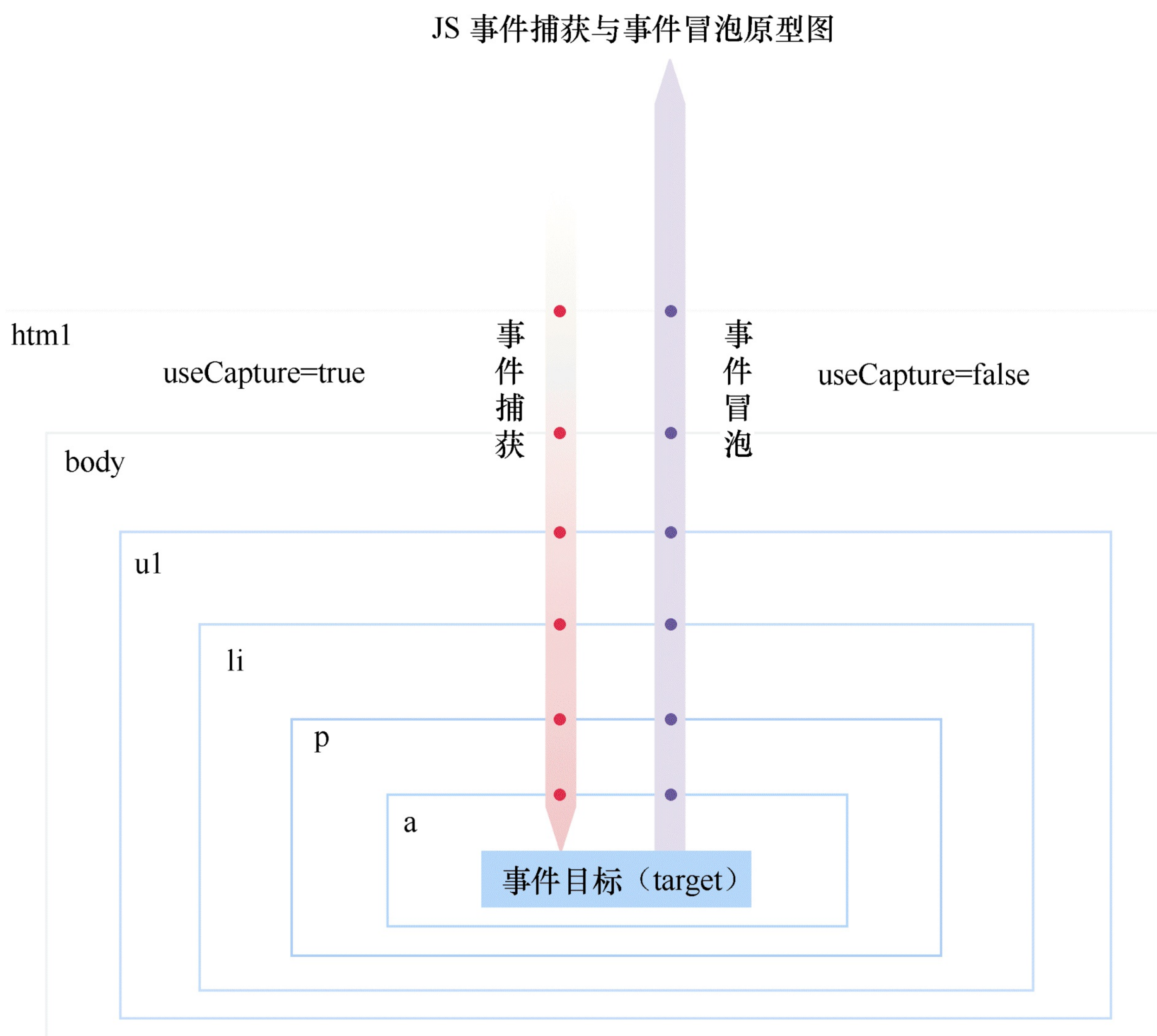


图10-2

不过IE还是不断进步的。

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html; cha
rset=UTF-8">
        <script>
```

```
            window.attachEvent("onload", function(e) {
                alert(e);
            });
        </script>
    </head>
    <body>
        <div>TODO write content</div>
    </body>
</html>
```

IE6、IE7的效果如图10-3所示。

IE8的效果如图10-4所示。

IE9的效果如图10-5所示。



图10-3

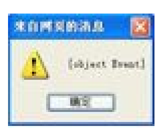


图10-4



图10-5

从一开始不知弹出什么东西，到现在明确这是一个事件对象，IE的确是在不断进步。不过，W3C的事件系统已经是大势所趋，微软也hold不住，从IE9起支持W3C那一套API，这对我们实现事件代理非常有帮助！

10.4　addEventListener的缺陷

W3C这一套API也不是至善至美，毕竟标准总是滞后于实现，剩下的那4个标准浏览器各有自己的算盘，它们之间亦有不一致的地方。

（1）新事件非常不稳定，可能还没有普及开就被废弃。在早期的Sizzle选择器引擎中，有这么几句：

```
document.addEventListener("DOMAttrModified", invalidate, false);
document.addEventListener("DOMNodeInserted", invalidate, false);
document.addEventListener("DOMNodeRemoved", invalidate, false);
```

现在这3个事件都被废弃了（准确来说，所有变动事件都完蛋了），FF14与Chrome18 开始用MutationObserver代替它。

（2）Firefox既不支持focusin、focus事件，也不支持DOMFocusIn、DOMFocusOut，直到现在也

不愿意用mousewheel代替DOMMouseScroll。Chrome不支持mouseenter与mouseleave。

因此不要以为标准浏览器就肯定会实现W3C钦定的标准事件，它们也有抗旨的时候，所以特征侦测必不可少。最可恨的是国内一些浏览器套用webkit内核，为了“超越”本版浏览器的HTML5跑分（HTML5test.com），竟然实现了一些无用的空接口来骗过特征侦测，因此必要时，我们还得使用非常麻烦的功能侦测来检测浏览器是否支持此事件。

（3）CSS3给私有实现添加自定前缀标识的坏习惯也漫延到一些与样式息息相关的事件名上。比如transitionend事件名，这个后缀名与大小写混合成5种形态，相当棘手。

（4）第3个参数useCapture 只有非常新的浏览器中才是可选项，比如FF6或之前版本是必选的，为安全起见，请确保第3个参数为布尔值。第4个参

数听说是FF专有实现，允许跨文档监听事件。第5个参数？的确存在第5参数，不过它只存在于Flash语言的同名方法中。现在前端工程师还是要求助于Flash，就作为一个知识点收下吧。有的面试官会跨界考这种东西。在Flash下，addEventListener的第4个参数用于设置该回调执行时的顺序，数字大的优先执行，第5个参数用来指定对侦听器函数的引用是弱引用还是正常引用。

（5）事件对象成员的不稳定。

W3C那套是从浏览器厂商抄来的，人家都用了这么久，难免与标准不一致。

FF下event.timeStamp返回0的问题，这个bug，2004年就有人提交了，直到2011年才被修复。

```
https://bugzilla.mozilla.org/show_bug.cgi?id=238041
```

Safari下event.target可能是返回文本节点。

event.defaultPrevented、event.isTrusted与stopImmediatePropagation的可用性很低，它们属于DOM3 event规范。

defaultPrevented属性是用于确定事件对象有没有调用 preventDefault方法。之前标准浏览器都统一用getPreventDefault方法来干这事，在jQuery源码中，你会发现它是用isDefaultPrevented方法来做。不过，isDefaultPrevented的确曾列入W3C草稿，可参见这里：

```
http://www.w3.org/TR/2003/WD-DOM-Level-3-Events-20030331/ecma-script-binding.html
```

isTrusted属性用于表示当前事件是否是由用户行为触发（比如说真实的鼠标点击触发一个click事件），还是由一个脚本生成的（使用事件构造方

法，比如event.initEvent）。

stopImmediatePropagation用于阻止当前事件的冒泡行为并且阻止当前事件所在元素上的所有相同类型事件的事件处理函数的继续执行。IE9+全部实现（但是，IE9、IE10的event.isTrusted有bug。link.click()后返回的也是true）。

Chrome5～Chrome17部分实现（event.isTrusted未支持）。Safari5才部分实现（event.isTrusted未支持）。

Opera10、Opera11部分实现（stopImmediatePropagation以及event.isTrusted未实现，而仅仅实现了defaultPrevented）。

Opera12部分实现（stopImmediatePropagation仍然未实现，但实现了e.isTrusted）。

Firefox1.0～Firefox5，未实现

stopImmediatePropagation和defaultPrevented，仅仅实现了event.isTrusted。isTrusted在成为标准前，是Firefox的私有实现。

Firefox6～Firefox10，仅未实现stopImmediatePropagation。Firefox11，终于实现了stopImmediate Propagation。

（6）标准浏览器没有办法模拟像IE6～IE8的propertychange事件。

虽然标准浏览器有input、DOMAttrModified、MutationObserver，但比起propertychange都弱爆了。propertychange可以监听多种属性变化，而不单单是value值，另外它不区分attribute和property，因此你无论是通过el.xxx = yyy，还是el.setAttribute(xxx, yyy)都接触过此事件。具体可参阅下面这篇博文。

```
http://www.cnblogs.com/rubylouvre/archive/2012/05/26/2519263.ht
```

```
ml
```

10.5　handleEvent与EventListenerOptions

浏览器是不断进化的，其他语言验证过的好东西会不断移植过来，像stopImmediatePropagation原来就是Flash的API。本节介绍的handleEvent也是出于Flash。

一般来说，我们是这样使用addEventListener的：

```
dom.addEventListener(type, handler, true/false)
```

handler是一个函数，浏览器会为它传入一个事件对象。

DOM2（**IE9**＋ 支持）开始支持另一种形式，handle为一个对象，里面有一个handleEvent方法。

```
var events = {
    attr: 222,
    handleEvent: function(event) {
          console.log(this.attr)//222
          switch (event.type) {
            case 'touchstart': this.touchstart(event); break;
            case 'touchmove': this.touchmove(event); break;
            case 'touchend': touchend(event); break;
         }
    },
    touchstart:function(event){
    },
    touchmove:function(event){
    },
    touchend:function(event){
    }
}

document.getElementById('elementID').addEventListener('touchsta
rt',events,false);
document.getElementById('elementID').addEventListener('touchmov
e',events,false);
document.getElementById('elementID').addEventListener('touchend
',events,false);
```

这样写有两个好处。

（1）将所有事件回调都集中一个对象上，方便共享信息。比如上面例子中的attr，由于this直接指向这个对象，我们就直接可以通过e.target.attr访问到它的值。

（2）动态改变事件处理器，通过e.type或其他信息调度相关的回调，不用先remove再add。

2016年，firefox开始支持DOM 4的**EventListenerOptions**：

```
dom.addEventListener(type, handler, EventListenerOptions);
```

EventListenerOptions里面有3个可选配置项，如果不写，默认都为false。

（1）capture，与之前一样，决定冒泡或捕获。

（2）passive，为true时，会将回调里面的e.preventDefault()操作无效化！

（3）once，为true时，此事件回调会只执行一次就自动解绑！

```
https://www.webreflection.co.uk/blog/2016/04/17/new-dom4-standa
rds
```

这个兼容性很差，Chrome49与Firefox49才开始支持。

10.6　Dean Edward“大神”的addEvent.js源码分析

这是Prototype时代早期出现的一个事件系统，jQuery事件系统的源头，亮点如下。

（1）有意识地屏蔽IE与W3C在阻止默认行为与事件传播的接口差异。

（2）处理IE执行回调时的顺序问题。

（3）处理IE的this的指向问题。

（4）没有平台检测代码，因为是使用最通用原始的onXXX构建。

（5）完全跨浏览器（包括IE4和NS4）。

（6）使用UUID来管理事件回调。

```
http://dean.edwards.name/weblog/2005/10/add-event/
```

```
function addEvent(element, type, handler) {
    //回调添加UUID，方便移除
    if (!handler.$$guid)
        handler.$$guid = addEvent.guid++;
    //元素添加events，保存所有类型的回调
    if (!element.events)
        element.events = {};
    var handlers = element.events[type];
    if (!handlers) {
        //创建一个子对象，保存当前类型的回调
        handlers = element.events[type] = {};
        //如果元素之前以onXXX = callback的方式绑定过事件，则成为当前类别第一个被触发的回调
        //问题是这回调没有UUID，只能通过el.onXXX = null移除
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }
    //保存当前的回调
    handlers[handler.$$guid] = handler;
    //所有回调统一交由handleEvent触发
    element["on" + type] = handleEvent;
}

addEvent.guid = 1;//UUID
//移除事件，只要从当前类别的储存对象delete就行
function removeEvent(element, type, handler) {
    if (element.events && element.events[type]) {
        delete element.events[type][handler.$$guid];
    }
}
function handleEvent(event) {
    var returnValue = true;
    //统一事件对象阻止默认行为与事件传统的接口
    event = event || fixEvent(window.event);
    //根据事件类型，取得要处理回调集合，由于UUID是纯数字，因此可以按照绑定时的顺序执行
    var handlers = this.events[event.type];
    for (var i in handlers) {
        this.$$handleEvent = handlers[i];
        //根据返回值判定是否阻止冒泡
        if (this.$$handleEvent(event) === false) {
            returnValue = false;
        }
    }
```

```
    return returnValue;
}

//对IE的事件对象做简单的修复
function fixEvent(event) {
    event.preventDefault = fixEvent.preventDefault;
    event.stopPropagation = fixEvent.stopPropagation;
    return event;
};
fixEvent.preventDefault = function() {
    this.returnValue = false;
};
fixEvent.stopPropagation = function() {
    this.cancelBubble = true;
};
```

不过在Dean Edward对应博文的评论中就可以看到许多指正与有用的patch。比如说，既然所有的修正都冲着IE去，那么标准浏览器直接使用addEventListener就行。有的还提到在iframe中点击事件时，事件对象不对的问题，提交以下有用补丁：

```
event = event || fixEvent(((this.ownerDocument || this.document
 || this).parentWindow || window).event);
```

其中第54条回复，直接导致了jQuery数据缓存

的系统的产生。为了避免交错引用产出的内存泄漏，建议元素就分配一个UUID，所有回调都放到一个对象中储存。

```
function addEvent(element, type, handler) {
    if (!handler.$$guid)
        handler.$$guid = addEvent.guid++;
    //每个元素都分配一个UUID，用于访问它们的所有回调
    if (!element.$$guid)
        element.$$guid = addEvent.guid++;
    if (!addEvent.handlers[element.$$guid])
        addEvent.handlers[element.$$guid] = {};
    //每个元素的回调都分类储存在不同的hash中
    var handlers = addEvent.handlers[element.$$guid][type];
    if (!handlers) {
        handlers = addEvent.handlers[element.$$guid][type] = {}
;
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }
    handlers[handler.$$guid] = handler;
    element["on" + type] = handleEvent;
}
addEvent.guid = 1;
// 放置所有回调的仓库
addEvent.handlers = {};
```

但随着时间的推移，使用者发现onXXX在IE存在不可消弥的内存泄漏。因此你翻看jQuery早期的版本，1.01版本是照抄Dean Edward的，从1.1.3.1版

本开始吸收上面提到的第54条回复的建议，元素只分配一个UUID，回调集中储放的方式了，并使用attachEvent/remove EventListener绑定事件——每个元素只绑定一次，然后所有回调都在类似handleEvent的函数中调用。不过这是后话，无损Dean Edward的光辉形象，他的addEvent在当时可是有着划时代的意义。无入侵式JavaScript只有在这样键壮的事件系统出现后才能得到迅猛发展。

10.7 jQuery的事件系统

jQuery的事件模块发端于Dean Edward的addEvent，然后它不断吸收社区的插件与补丁，发展成为非常成熟的事件系统。其中不得不提的是其事件代理与事件派发机制。

早在2007年，Brandon Aaron为jQuery写了一个划时代的插件叫livequery，它可以监听后来插入的元素的事件。比如，一个表格，我们为tr元素绑定mouseover/mouseout事件时，只有十行，然后我们又动态添加了20行，这20个tr元素也同样能执行mouseover/mouseout回调。魔术在于，它并没有把事件侦探器绑定在tr元素上，而是绑定最顶层的document上，然后通过事件冒泡，取得事件源，判定它是否匹配用户给定的CSS表达式，才执行用户回调。

如果一个表格有100个tr元素，每个都要绑定mouseover/mouseout事件，改成事件代理的方式，可以节省99次绑定，这优化很好，更何况它能监听将来添加的元素，因此立马被吸收到jQuery1.3中去，成为它的live方法。再把一些明显的bug修复了，jQuery1.32成为受欢迎的版本，与后来的jQuery1.42都是里程碑式！不过话说回来，live方法需要对某些不冒泡的事件做些处理，比如一些表单事件，有的只冒泡到form，有的冒泡到document，有的压根不冒，如表10-1所示。

表**10-1**

	IE6	**IE8**	**FF3.6**	**Opera10**	**Chrome4**	**Safari4**
submit	form	form	document	document	document	document
reset	form	form	document	document	form	form
change	不冒泡	不冒泡	document	document	不冒泡	不冒泡
click	document	document	document	document	document	document

select	不冒泡	不冒泡	document	document	不冒泡	不冒泡

对于focus、blur、change、submit、reset、select等不会冒泡的事件，在标准浏览器中，我们可以设置addEventListener的最后一个参数为true轻松搞定。IE就有点麻烦了，要用focusin代替focus，focusout代替blur，selectstart代替select。change、submit与reset就复杂了，必须利用其他事件来模拟，还要判断事件源的类型，selectedIndex、keyCode等相关属性，这课题最后由一个叫reglib的库搞定。reglib的作者还写了一篇很著名的博文《Goodbye mouseover，hello mouseenter》，来推广微软系的两个事件mouseenter与mouseleave。正如你们今天看到那样，jQuery全面接纳了它们。

live方法带来的全新体验是空前的，但毕竟要冒泡到最顶层，对IE还是有点坎坷，有时还会失灵。最好能指定父节点，一个绑定时已经存在的父

节点，这样就不用费力了。当时有3篇博文提出相近的方案，它们给出的接口一篇比一篇接近John Resig接纳的方案。

比如最后那篇博文，它已经是这个样子：

```
$("div").delegate( 'click', 'span', function( event ){
        $( this ).toggleClass('selected');
        return false;
});
```

并提出对应解除代理的API——undelegate。而jQuery1.42在2010年2月19日推出时，也是这两个接口，前两个参数只调换一下：

```
$("div").delegate( "span","click", function( event ){
        $( this ).toggleClass('selected');
        return false;
});
```

正所谓“众人拾柴火焰高”，jQuery的强大不无道理。在jQuery 1.8中，它又吸收dperini / nwevents

的点子，改进其事件代理，大大提高代理性能。

先看一下其主要接口，如图10-6所示。

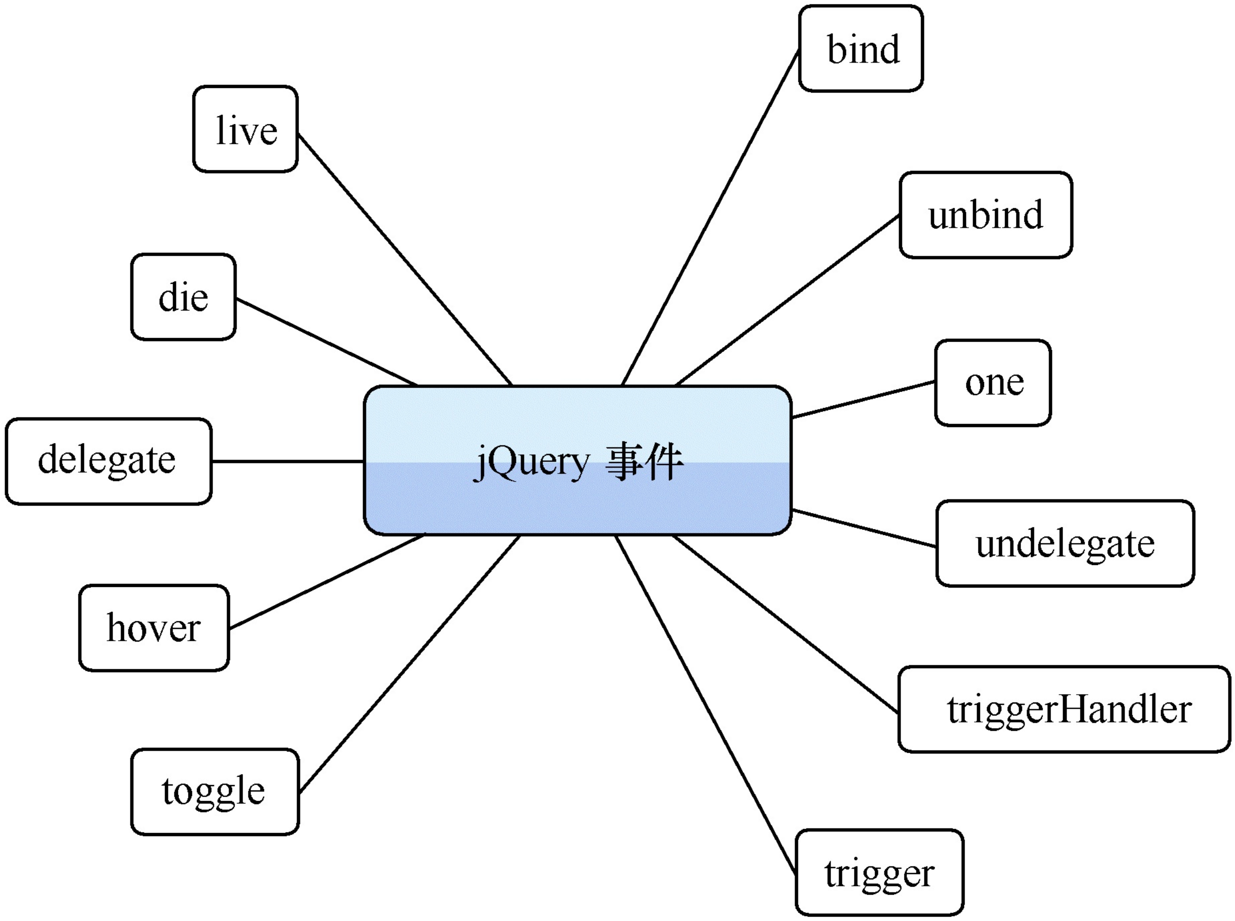


图10-6

其中bind、unbind、one、trigger、toggle、hover、ready一开始就有。

tirggerHandler是jQuery 1.23增加的，内部依赖于trigger，只对当前匹配元素的第一个有效，不冒泡不触发默认行为。

live与die是jQuery 1.3增加的，用于事件代理，统一由document代理。

delegate与undelegate是jQuery 1.42增加的，允许指定代理元素的事件代理，它内部是利用live，die实现的。

on与off是jQuery 1.7增加的，目的是统一事件接口。bind、one、live、delegate直接由on衍生，unbind、die、undelegate直接由off衍生。

hover用于模拟css的hover效果，内部依赖于mouseenter与mouseleave。

ready可以看作为load事件的增强版，获取最早的DOM可用时机后立即执行各种DOM操作。

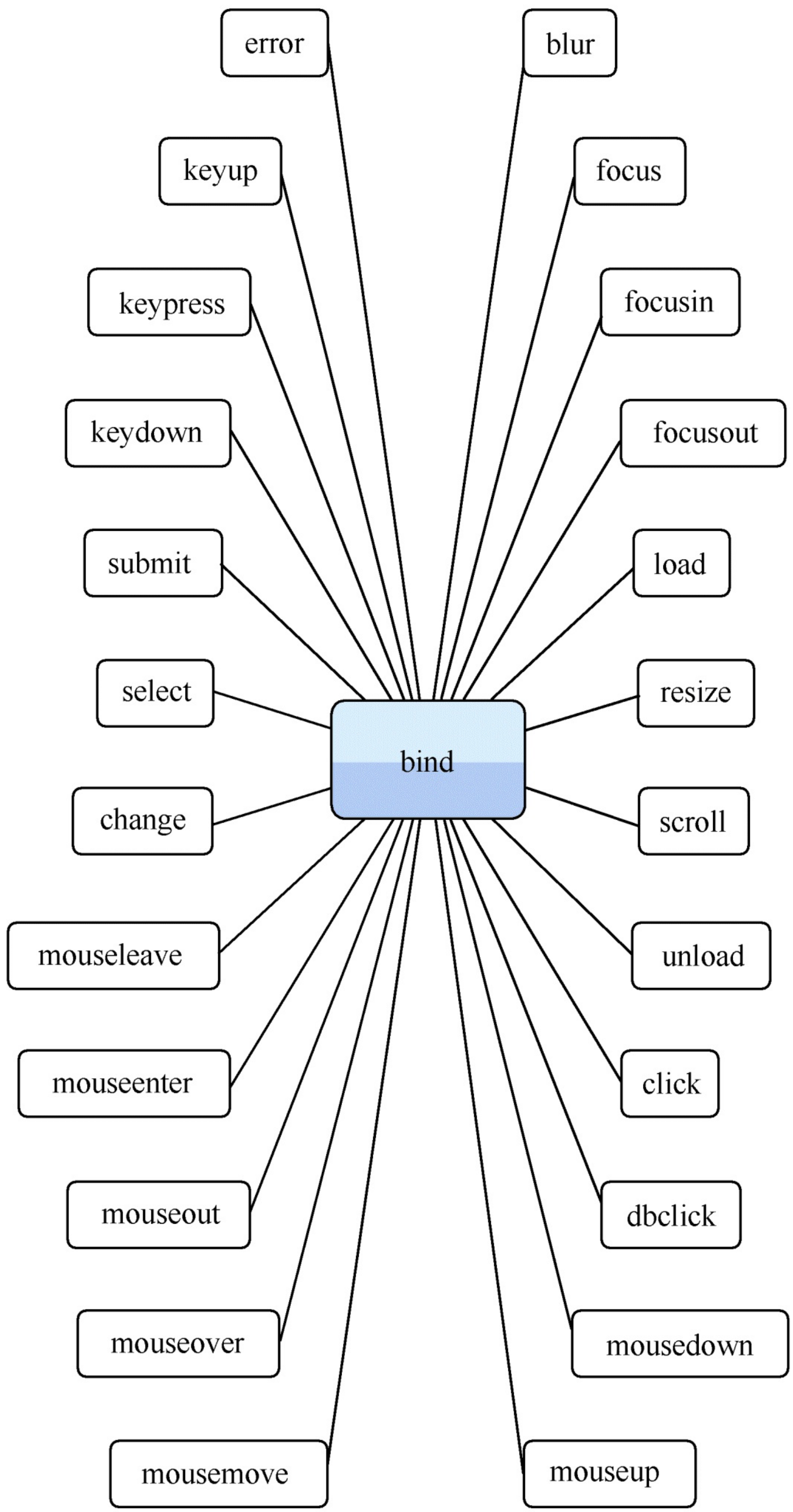
toggle是click的加强版，每次点击都执行不同的回调并切换到下一个（jQuery 2.0被废弃）。

trigger与triggerHandler是jQuery的fireEvent实现。

此外，jQuery还有25个以事件类型命名的快捷方法，当传参数个数为2时表现为绑定事件，个数为0表现为派发事件，如图10-7所示。

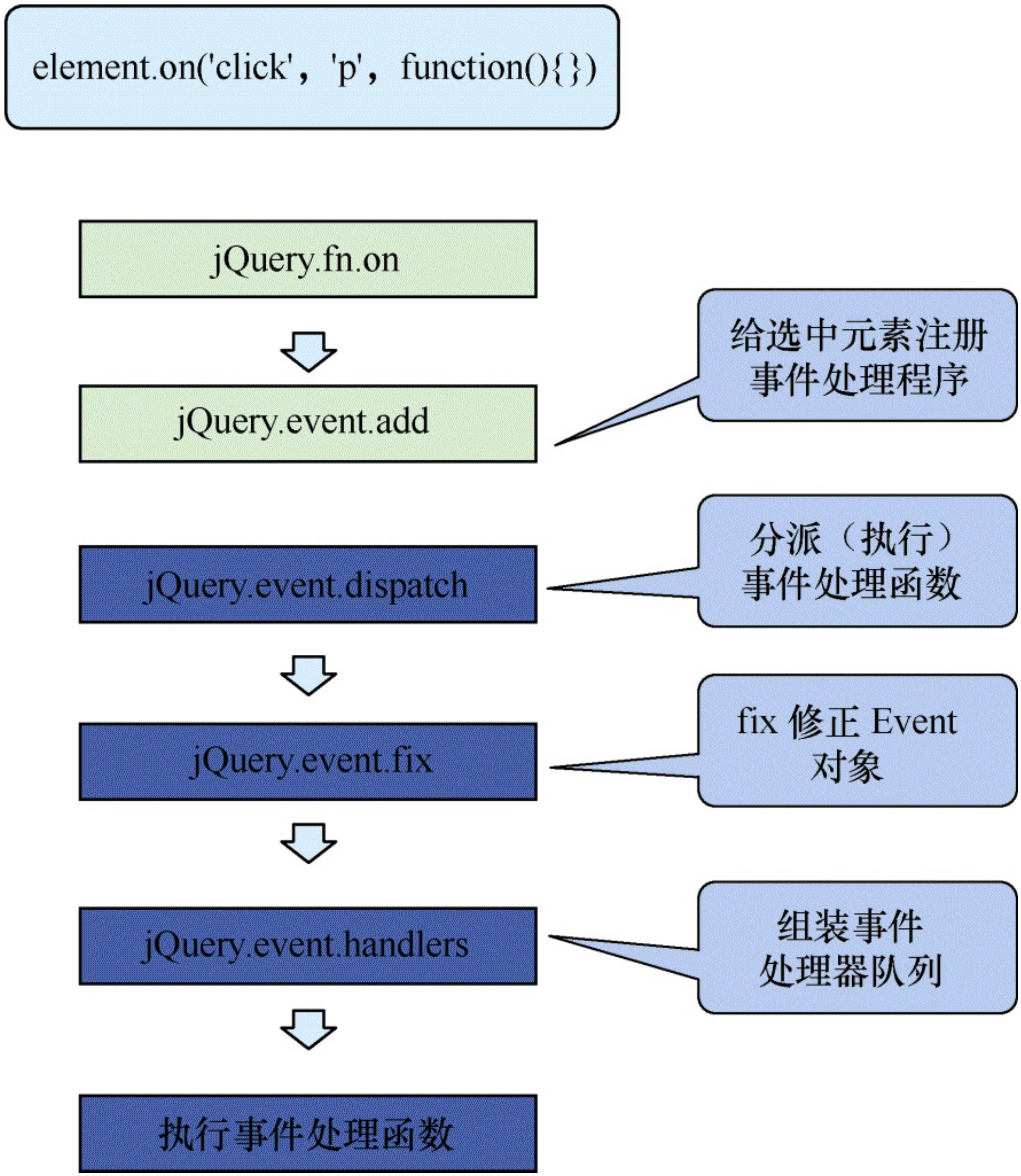
图10-8是jQuery的事件流程，大家可以稍微了解一下。

jQuery的事件系统是前MVVM时代，最强大的事件系统，具有高扩展性、高性能、功能丰富的优点。这个直到React的事件系统出现才被超越，MVVM时代的事件系统已经倾向于内部使用事件代理来绑定一切事件回调，从而获得更高的性能。



error
blur
keyup
focus
keypress
focusin
keydown
focusout
submit
load
select
resize
bind
change
scroll
mouseleave
unload
mouseenter
click
mouseout
dbclick
mouseover
mousedown
mousemove
mouseup

图10-7



element.on('click'，'p'，function(){})
jQuery.fn.on
jQuery.event.add
给选中元素注册事件处理程序
jQuery.event.dispatch
分派（执行）事件处理函数
jQuery.event.fix
fix 修正 Event 对象
jQuery.event.handlers
组装事件处理器队列
执行事件处理函数

图10-8

10.8　avalon2的事件系统

avalon2的事件系统最代表未来的发展方向，企图通过事件代理绑定一切可能的事件。它的两个基础方法与jQuery一样，分别叫做bind、unbind，里面也采取之前cssHooks、valHooks一样的钩子对象来对付兼容性问题。但不同的是，它还使用了缓存机制。在avalon中没有实现专门的数据缓存系统，因此这个缓存系统只是一个普通的javascript对象，但够用就可以了。此外，还有UUID系统，但只用于元素节点，我们是通过setAttribute来添加的，因此没有对付ember、object、avalon的事件系统主要由avalon.bind、avalon.ubind这两个方法组成。先说avalon.bind，它与业界常见的addEvent方法一样，拥有3个参数。

（1）el：绑定事件的对象。

（2）type：事件的类型，可以是自定义事件。

（3）fn：事件回调，第一个参数总是事件对象，并且拥有type与target属性。

然后avalon拥有3个专门辅助事件系统的对象（或叫哈希），用于修正最初传入的事件名，回调及存放地点。

（1）bubbleEvents：存储所有能冒泡的事件类型，并且针对不同浏览器有所增删，键名为事件名，值为true。

（2）eventHooks:：用于处理特殊事件，键名为事件名，值为一个钩子函数，要求将初始参数传进去，最后返回一个新回调。比如滚轮，mouseenter/leave等难缠的事件就需要在钩子方法中处理。

（3）eventListeners：存放回调函数，如果原

事件是命中eventHooks，那么存放的是新回调。键名是uuid，它位于回调的上面。

然后将事件名＋uuid存放到元素节点的avalon-events属性中，以逗号隔开。如果此事件能冒泡，那么也在html的delegate-events属性值中加上对应事件名，如图10-9所示。

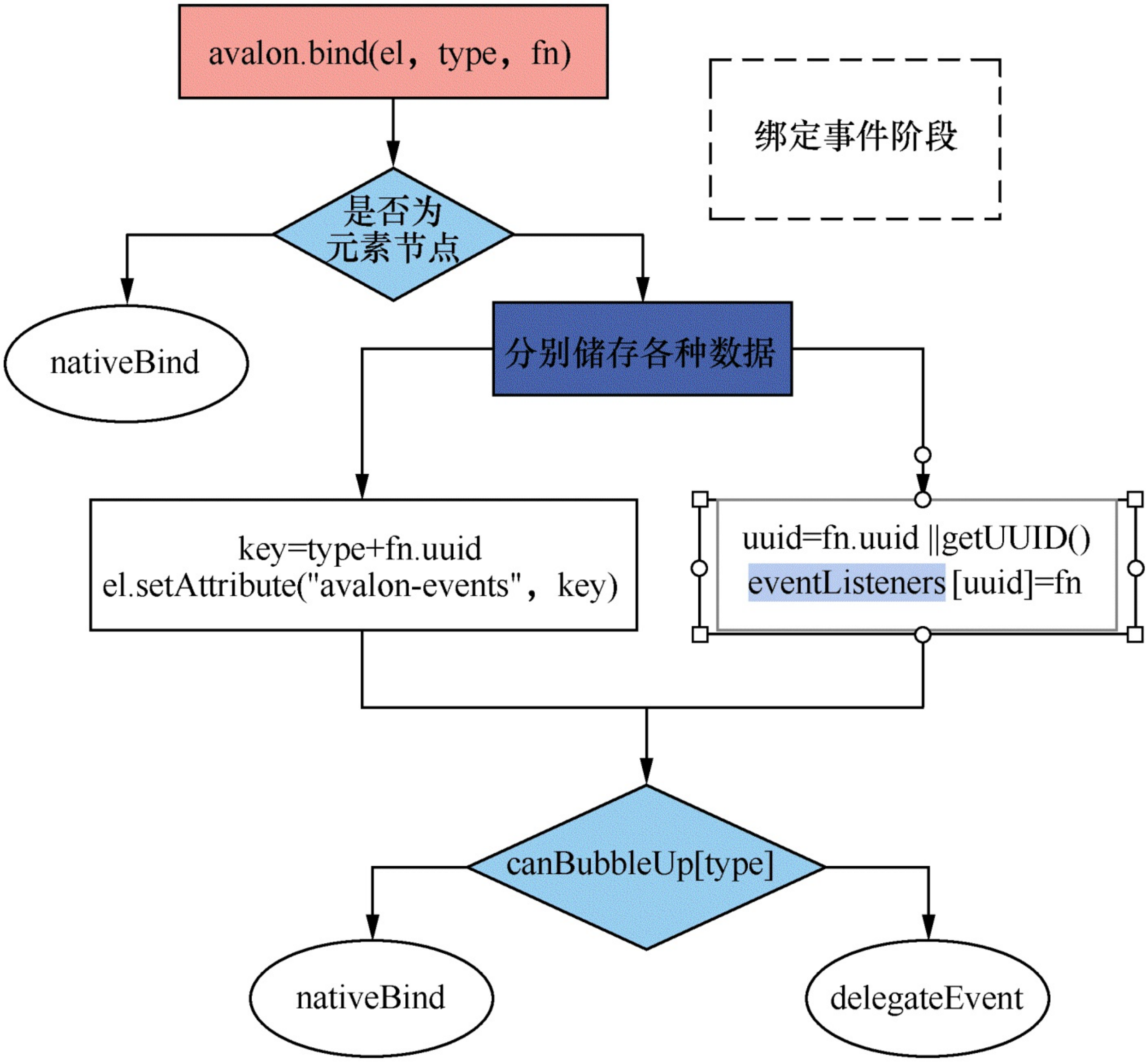


图10-9

最后根据事件能否冒泡，决定将回调绑定到根节点还是当前元素上。

```
var canBubbleUp = avalon.bubbleEvents = require('./bubbleEvents')
```

```
if (!W3C) {
    delete canBubbleUp.change
    delete canBubbleUp.select
}
var uuid = 1
var eventHooks = avalon.eventHooks
var focusBlur = {
    focus: true,
    blur: true
}
/*绑定事件*/
avalon.bind = function (elem, type, fn) {
    if (elem.nodeType === 1) {
        var value = elem.getAttribute('avalon-events') || ''
        var uuid = fn.uuid || (fn.uuid = ++uuid)
        var hook = eventHooks[type]
        if(hook){
            type = hook.type || type
            if (hook.fix) {
                fn = hook.fix(elem, fn)
                fn.uuid = uuid
            }
        }
        var key = type + ':' + uuid
        avalon.eventListeners[fn.uuid] = fn
        if (value.indexOf(type + ':') === -1) {
        //同一种事件只绑定一次
        //对于能冒泡的事件，或现代浏览器下的focus，blur事件使用事件代理
        //因为现代浏览器可以使用事件捕获方法监听事件
            if (canBubbleUp[type] || (avalon.modern && focusBlu
r[type])) {
                delegateEvent(type)
            } else {
                nativeBind(elem, type, dispatch)
            }
        }
        var keys = value.split(',')
        if (keys[0] === '') {
            keys.shift()
        }
        if (keys.indexOf(key) === -1) {
            keys.push(key)
            elem.setAttribute('avalon-events', keys.join(','))
            //将令牌放进avalon-events属性中
```

```
        }

    } else {
        nativeBind(elem, type, fn)
    }
    return fn //兼容之前的版本
}
```

nativeBind就是本章最开始的addEvent，只是对attachEvent与addEventListener进行简单的封装。

```
function delegateEvent(type) {
    var value = root.getAttribute('delegate-events') || ''
    if (value.indexOf(type) === -1) {
        var arr = value.match(avalon.rword) || []
        arr.push(type)
        root.setAttribute('delegate-events', arr.join(','))
        nativeBind(root, type, dispatch, !!focusBlur[type])
    }
}
```

delegeteEvent内部也是调用dispatch事件，只是把绑定地点换成根节点。

dispatch事件做了3件事件，解决兼容性问题。

（1）对原生事件对象进行再封装，解决事件

对象的方法名与属性名不一致问题，并且针对某些特殊事件（比如IE有Chrome没有的`mouseenter/leave`事件和Firefox不支持的`mousewheel`事件），我们在这里 事件冒充 （用mouseover事件冒充mouseenter事件）

（2）通过event.target得到事件源对象，从而访问到avalon-events属性值，然后根据event.type得到所有相对UUID，再从eventListeners得到回调。这些步骤通过collectHandlers私有方法处理。

（3）遍历所有回调，根据事件冒泡与事件返回值，设置一些分支，实现stopPropagation与stopImmediatePropagation（如图10-10和图10-11所示）。

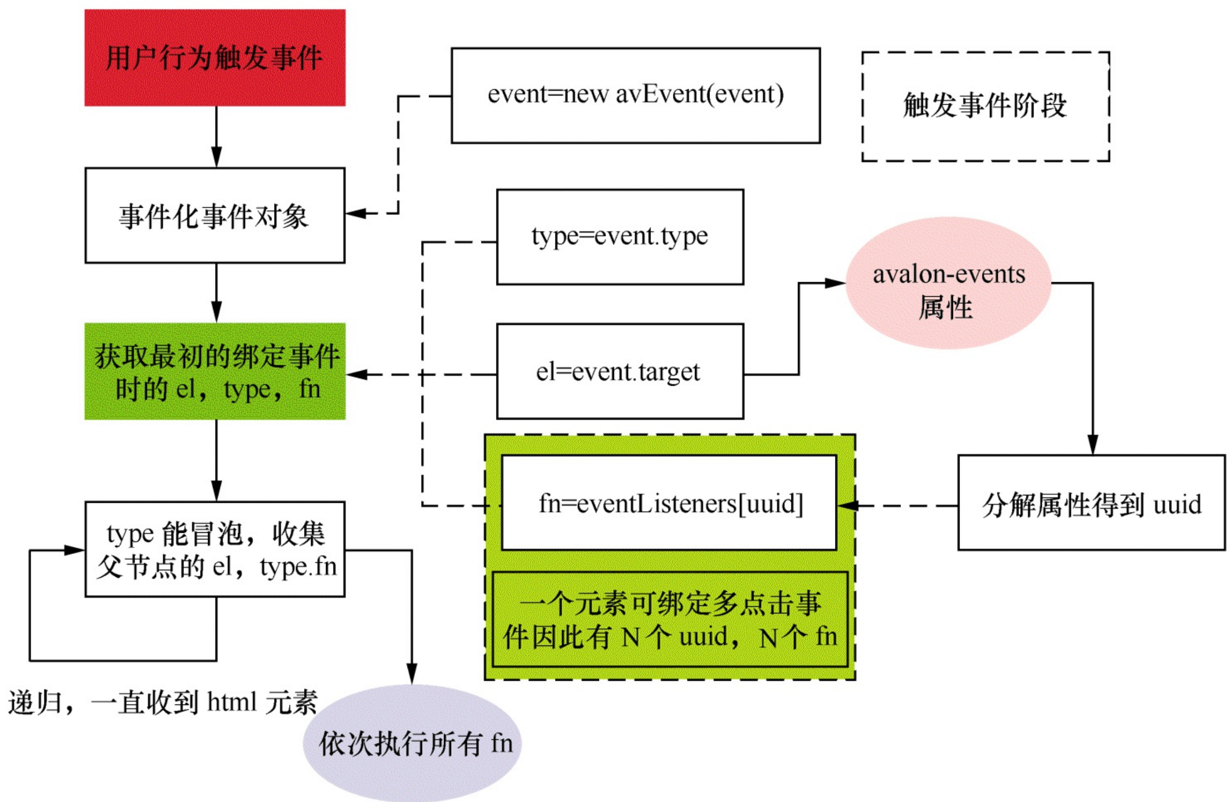
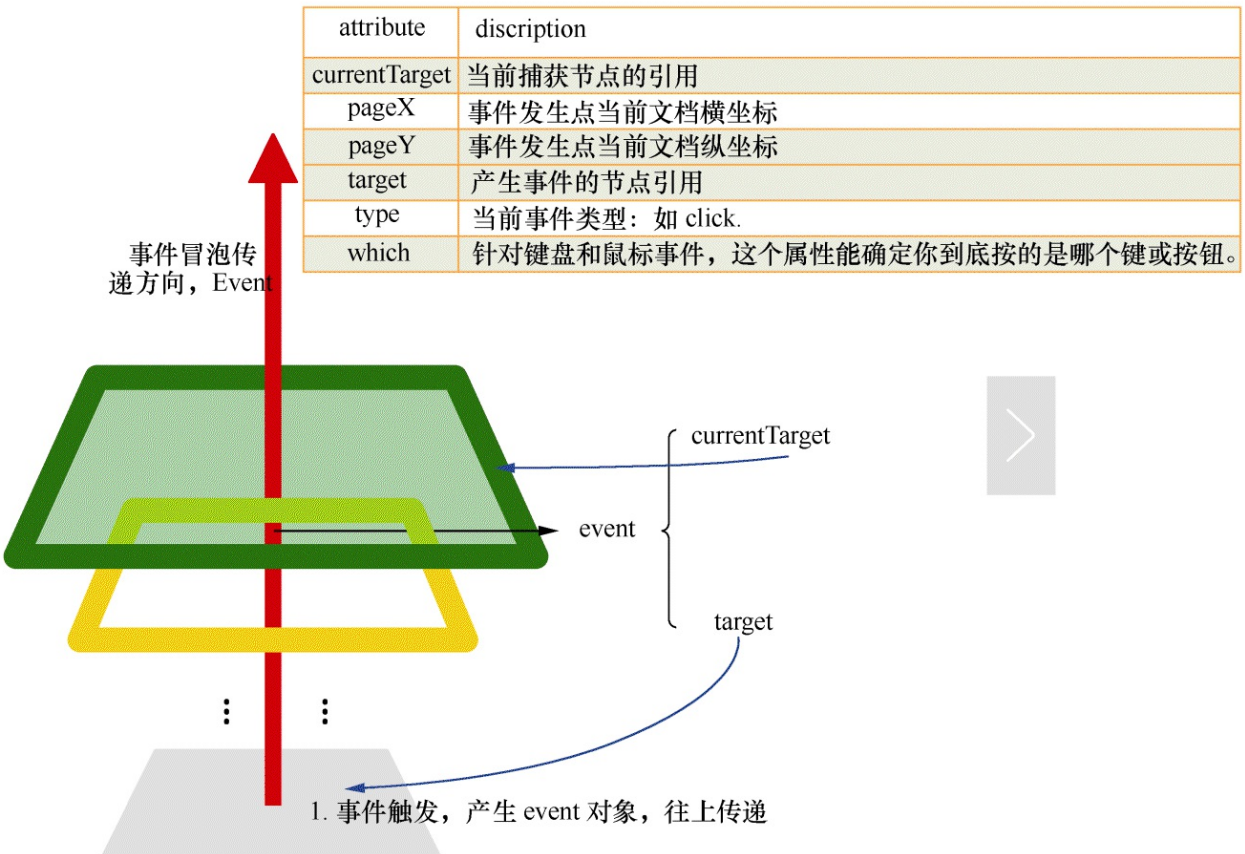


图10-10

```
var typeRegExp = {}
function collectHandlers(elem, type, handlers) {
    var value = elem.getAttribute('avalon-events')
    if (value && (elem.disabled !== true || type !== 'click'))
{
        var uuids = []
        var reg = typeRegExp[type] || (typeRegExp[type] = new R
egExp("\\b"+type + '\\:([^,\\s]+)', 'g'))
        value.replace(reg, function (a, b) {
            uuids.push(b)
            return a
        })
        if (uuids.length) {
            handlers.push({
                elem: elem,
                uuids: uuids
            })
        }
```

```
    }
    elem = elem.parentNode
    if (elem && elem.getAttribute && canBubbleUp[type]) {
        collectHandlers(elem, type, handlers)
    }
}
function dispatch(event) {
    event = new avEvent(event)
    var type = event.type
    var elem = event.target
    var handlers = []
    collectHandlers(elem, type, handlers)
    var i = 0, j, uuid, handler
    //这里会模拟整个冒泡过程及当用户在回调里调用了stopPropagation与stop
ImmediatePropagation //方法时，会改变事件的cancelBubble、isImmediat
ePropagationStopped属性，如何中断冒泡的
    while ((handler = handlers[i++]) && !event.cancelBubble) {
        var host = event.currentTarget = handler.elem
        j = 0
        while ((uuid = handler.uuids[ j++ ]) &&
                !event.isImmediatePropagationStopped) {

            var fn = avalon.eventListeners[uuid]
            if (fn) {
                var ret = fn.call( elem, event)
                if(ret === false){
                    event.preventDefault()
                    event.stopPropagation()
                }
            }
        }
    }
}
```



attribute	discription
currentTarget	当前捕获节点的引用
pageX	事件发生点当前文档横坐标
pageY	事件发生点当前文档纵坐标
target	产生事件的节点引用
type	当前事件类型：如 click.
which	针对键盘和鼠标事件，这个属性能确定你到底按的是哪个键或按钮。

图10-11

至于怎样标准化事件对象，所有框架的想法都趋于一致，就是将srcElement改成target并修正停止事件传播与阻止默认行为这两个API，加上timeStamp与originalEvent属性。像Opera与Chrome，它们是同时支持两套API。

```
function avEvent(event) {
    if (event.originalEvent) {
```

```
        return this
    }
    for (var i in event) {
        if (!(ypeof event[i] !== 'function') ){
            this[i] = event[i]//不复制事件
        }
    }
    if (!this.target) {
        this.target = event.srcElement
    }
    this.timeStamp = new Date() - 0
    this.originalEvent = event
}
avEvent.prototype = {
    preventDefault: function () {
        var e = this.originalEvent
        this.returnValue = false
        if (e) {
            e.returnValue = false
            if (e.preventDefault) {
                e.preventDefault()
            }
        }
    },
    stopPropagation: function () {
        var e = this.originalEvent
        this.cancelBubble = true
        if (e) {
            e.cancelBubble = true
            if (e.stopPropagation) {
                e.stopPropagation()
            }
        }
    },
    stopImmediatePropagation: function () {
        var e = this.originalEvent
        this.isImmediatePropagationStopped = true
        if (e.stopImmediatePropagation) {
            e.stopImmediatePropagation()
        }
        this.stopPropagation()
    }
}
```

我们再看一下如何删除事件。由于我们的事件名与回调都可以在avalon-events属性值中得到，主要删掉里面的UUID，那么当点击事件发生时，就不会进入我们的用户回调中。同理，如果将事件名删掉，那么这个元素的所有点击回调就不会触发。进一步，我们把avalon-events这个属性都删掉，所有事件回调就不会触发。当然为节约内存起见，最好也把avalon.eventListeners中的回调也删掉，这样不用操作detachEvent/removeEventListeners就能卸载事件。

```
avalon.unbind = function (elem, type, fn) {
    if (elem.nodeType === 1) {
        var value = elem.getAttribute('avalon-events') || ''
        switch (arguments.length) {
            case 1:
                nativeUnBind(elem, type, dispatch)
                elem.removeAttribute('avalon-events')
                break
            case 2:
                value = value.split(',').filter(function (str)
{
                    return str.indexOf(type + ':') === -1
                }).join(',')

                elem.setAttribute('avalon-events', value)
                break
            default:
                var search = type + ':' + fn.uuid
```

```
                value = value.split(',').filter(function (str)
{
                    return str !== search
                }).join(',')
                elem.setAttribute('avalon-events', value)
                delete avalon.eventListeners[fn.uuid]
                break
        }
    } else {
        nativeUnBind(elem, type, fn)
    }
}
```

10.9　总结

事件是我们与页面交互的重要渠道，因此浏览器提供了琳琅满目的事件给我们使用，熟练掌握各种事件是一个高级JSer的标志。虽然事件存在巨大的差异性，但经过先人的奋斗，最终提炼出了现在这种基于UUID的事件系统解决方案，而为了提高性能，又发明了事件代理这套机制。UUID＋事件代理，必将成为MVVM实现事件绑定的主流方式。

最后，重要的事说3次。事件系统必须有UUID！必须有UUID！必须有UUID！有了UUID，以后手动触发事件或删除事件，就可以绕过浏览器的那些繁杂琐碎的API了。由于浏览器的删除事件必须传入原来的回调，因此对不注意保存引用的普通前端来说，这个以后删除几乎不可能。而手动触发事件，光是几十initXXXEvent，createEvent的组合就搞晕你了。这也是本章直接忽略如何手动触发

事件的原因。

第11章　移动端的事件系统



移动端事件不同于PC端，许多PC端事件在移动端是不可用的，需要综合多个原始事件模拟出某一个可用的事件。究其原因，是由于最早在移动端发迹的iOS给我们带来各种各样的奇葩bug或怪异设定，安卓也无脑照抄但又抄不好，以致于带来更多bug。

最大的难题：触屏交互300ms延迟。在2007年苹果准备发布首款iPhone时，他们的工程师需要解决如何让用户在手机上浏览那些为PC大屏幕设备而设计的网页，那时候的网页并没有响应式设计，并且都是使用px作为字体和元素尺寸的单位。因此用户在使用iPhone浏览网页时体验非常糟糕，不是字体过小，就是页面只显示一部分。于是苹果工程师想到了一个办法，那就是如今大家习以为常的双击放大页面，再次双击恢复页面功能。有了这个设计之后，PC端的dblclick事件被废除掉了。

网页中难免有大量链接，如果我们在移动设备中访问网页时点击到某个链接，如何区分用户是想打开该链接还是放大页面呢？而且移动设备屏幕较小，容易出现误操作。因此苹果工程师就想到让所有的点击事件延迟一下，以判断用户是否再次点击屏幕来判断用户的操作。不过，这个处理方式只是为了方便开发者，完全没有考虑到用户的感受。并

且随着响应式设置的流行，我们完全可以在设计之初就避免上述误点击的问题。因此，旧有的点击延迟便成为众矢之的，第一个针对此问题的库就是faskclick。

此外，智能手机上的交互基本上是通过手指^touch进行的，要用一个手指或多个手指的动作来表述自己的意见，显然不能只靠点击，于是出现了划动（swipe）、拖动（drag）、缩放（pinch）。由于点击事件不可用，于是人们搞出一个同义事件tap。本章主要围绕这几种事件展开。

avalon的事件系统基本上是以iOS手势事件为蓝本改进而来. 其中由于tap事件需要大费周章的模拟PC上的点击效果，因此独立出来。

本章介绍到的源代码都可以在以下链接找到：

```
https://github.com/RubyLouvre/avalon2/tree/master/src
https://github.com/RubyLouvre/avalon2/tree/master/component/gesture
```

11.1　touch系事件

在移动端上许多事件都是使用这两种事件合成出来的，包含iOS2.0软件的iPhone 3G发布时，也包含了一个新版本的Safari浏览器。

该移动版Safari提供了一些与触摸（touch）操作相关的新事件。后来，Android上的浏览器也实现了相同的事件。触摸事件会在用户手指放在屏幕上面时、在屏幕上滑动时或从屏幕上移开时触发。具体来说，有以下几个触摸事件。

- touchstart：一根或多根手指开始触摸屏幕时触发。
- touchmove：一根或多根手指在屏幕上移动时触发。在这个事件发生期间，调用preventDefault()可阻止滚动及文字选中。
- touchend：当手指从屏幕上移开时触发。

- touchcancel：触摸意外取消时触发。比如你一直长按不动，或者手指划出屏幕，或者触摸过程中打入电话，系统就会自动调度此事件。

上面这几个事件都会冒泡，也都可以取消。虽然这些触摸事件没有在DOM规范中定义，但它们却是以兼容DOM的方式实现的。因此，每个触摸事件的event对象都提供了在鼠标事件中常见的属性：bubbles、cancelable、view、detail、altKey、shiftKey、ctrlKey和metaKey，但你会发现所有表示坐标的属性不见了，如图11-1所示。

这些坐标属性被放到一个叫Touch的对象上，而它们又分属于touches、targetTouches、changeTouches对象上。

- touches：当前屏幕上所有触摸点列表。
- targetTouches：绑定事件的元素节点或对象上的所有触摸点列表。

- changeTouches：绑定事件的元素节点或对象上发生了改变的触摸点列表。

每个Touch对象包含下列属性。

- clientX：触摸目标在视口中的x坐标。
- clientY：触摸目标在视口中的y坐标。
- identifier：表示触摸的唯一ID。
- pageX：触摸目标在页面中的x坐标。
- pageY：触摸目标在页面中的y坐标。
- screenX：触摸目标在屏幕中的x坐标。
- screenY：触摸目标在屏幕中的y坐标。
- target：触摸的DOM节点坐标。
- 在一些新浏览器，Touch对象甚至包括了它们的移动距离，旋转角度如图11-2所示。

```
▼ TouchEvent {}
    altKey: false
    bubbles: true
    cancelBubble: false
    cancelable: true
  ▶ changedTouches: TouchList
    charCode: 0
    ctrlKey: false
    currentTarget: null
    defaultPrevented: false
    detail: 0
    eventPhase: 0
    keyCode: 0
    metaKey: false
  ▶ path: Array[6]
    returnValue: true
    shiftKey: false
  ▶ srcElement: div.doubletap
  ▶ target: div.doubletap
  ▶ targetTouches: TouchList
    timeStamp: 1444024572139
  ▶ touches: TouchList
    type: "touchstart"
  ▶ view: Window
    which: 0
  ▶ __proto__: TouchEvent
```

图11-1

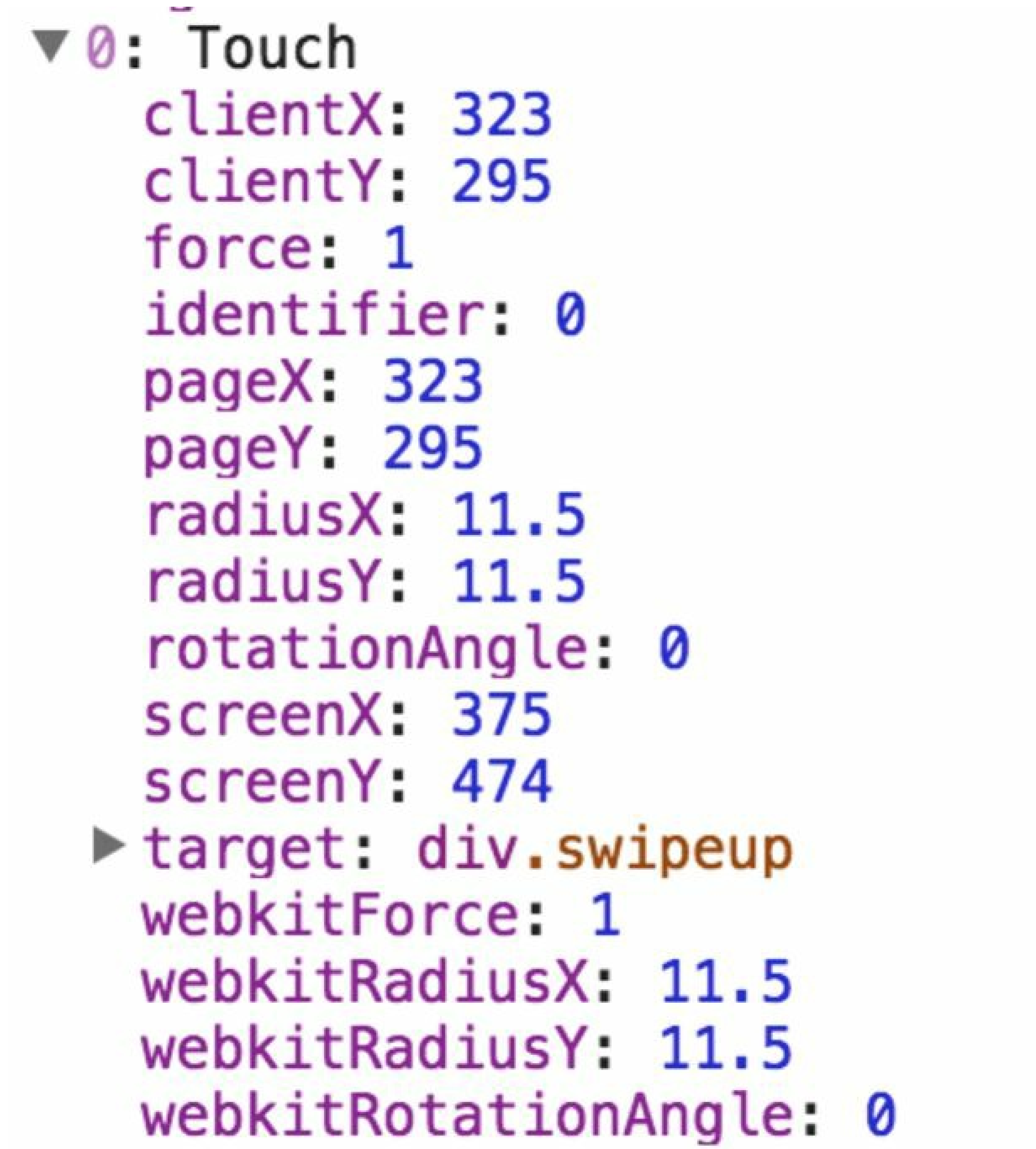

0: Touch
clientX: 323
clientY: 295
force: 1
identifier: 0
pageX: 323
pageY: 295
radiusX: 11.5
radiusY: 11.5
rotationAngle: 0
screenX: 375
screenY: 474
target: div.swipeup
webkitForce: 1
webkitRadiusX: 11.5
webkitRadiusY: 11.5
webkitRotationAngle: 0

图11-2

值得注意的是，如果我们想获取触摸点的信息，最好一直使用changeTouches，并且在touchend事件中touches及targetTouches的长度为零。

从行为上来看，touch事件与鼠标事件非常相似，当我们点击某一个元素时，它们几乎都会同时发生，其执行顺序如下。

（1）touchstart。

（2）mouseover。

（3）mousemove。

（4）mousedown。

（5）mouseup。

（6）click。

（7）touchend。

当然顺序不一定对的，移动端的兼容性问题比IE6更让人头痛，我只能保证touchstart、mousedown、mouseup、click这4个的顺序。在一些不支持touch事件的浏览器上，有的库想用鼠标事件来模拟各种触屏事件，其实这样做非常不好。首先鼠标事件响应非常慢，其次无法摸拟多点触摸。

网上提供关于touch事件与鼠标事件的比较，如图11-3所示。

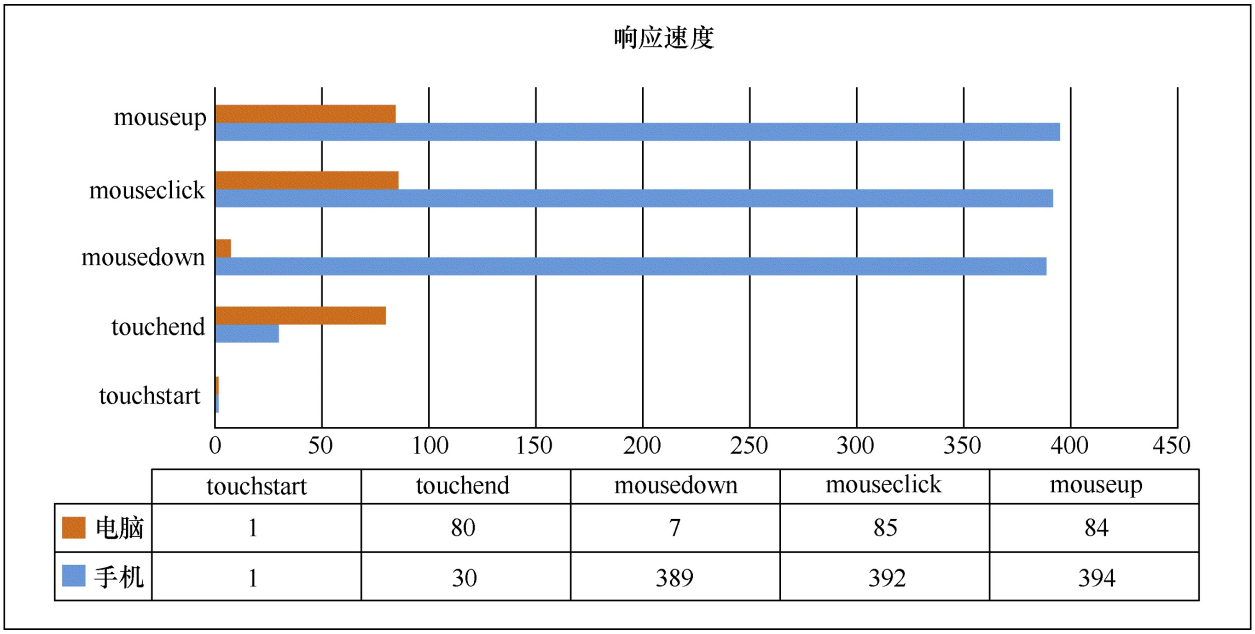


图11-3

其实不支持touch事件的浏览器也只有winphone下的IE而已，我们可以通过表11-1所示的两组事件与touch事件对应起来。

表**11-1**

标准事件名	**touchstart**	**touchmove**	**touchend**	**touchcancel**
IE9-10	MSPointerDown	MSPointerMove	MSPointerUp	MSPointerCancel
IE11	pointerdown	pointermove	pointerup	pointercancel
模拟事件	mousedown	mousemove	mouseup	

11.2　gesture系事件

iOS 2.0中的Safari还引入了一组手势事件。当两个手指触摸屏幕时就会产生手势，手势通常会改变显示项的大小，或者旋转显示项。有3个手势事件，分别如下。

（1）**gesturestart**：当一个手指已经按在屏幕上面时，另一个手指也触摸屏幕时触发。

（2）**gesturechange**：当触摸屏幕的任何一个手指的位置发生变化时，触发。

（3）**gestureend**：当任何一个手指从屏幕上面移开时，触发。

只有两个手指都触摸到屏膜的接收容器时才会触发这些事件。在一个元素上设置事件处理程序，意味着两个手指必须同时位于该元素的范围之内，

才能触发手势事件（这个元素就是目标）。由于这些事件冒泡，所以将事件处理程序放在文档上也可以处理所有手势事件。此时，事件的目标就是两个手指都位于其范围内的那个元素。

触摸事件和手势事件之间存在某种关系。当一个手指放在屏幕上时，会触发touchstart事件。如果另一个手指又放在了屏幕上，则会先触发gesturestart事件。如果另一个手指又放在了屏幕上，则会先触发gesturestart事件，随后触发基于该手指的touchstart事件。如果一个或两个手指在屏幕上滑动，将会触发gesturechange事件，但只要有一个手指移开，就会触发gestureend事件，紧接着又会触发基于该手指的touchend事件。

与触摸事件一样，每个手势事件的event对象都包含着标准的鼠标事件属性：bubbles、cancelable、view、clientX、clientY、screenX、

screenY、detail、altKey、shiftKey、ctrlKey和metaKey。此外，还包含两个额外的属性：rotation和scale。其中，rotation属性表示手指变化引起的旋转角度，负值表示逆时针旋转，正值表示顺时针旋转（该值从0开始）。而scale属性表示两个手指间距的变化情况（例如向内收缩会缩短距离），这个值从1开始，并随距离拉大而增长，随距离缩减而减小。

下面是使用手势事件的一个示例。

```
function handleGestureEvent(event) {
    var output = document.getElementById("output");
    switch(event.type) {
        case "gesturestart":
            output.innerHTML = "Gesture started (rotation=" + e
vent.ratation +",scale="
            + event.scale + ")";
            break;
        case "gestureend":
            output.innerHTML += "<br>Gesture ended (rotation+"
+ event.rotation +
            ",scale=" + event.scale + ")";
            break;
        case "gesturechange":
            output.innerHTML += "<br>Gesture changed (rotation+
=" + event.rotation +
            ",scale+" + event.scale + ")";
            break;
    }
```

```
}
document.addEventListener("gesturestart", handleGestureEvent, f
alse);
document.addEventListener("gestureend", handleGestureEvent, fal
se);
document.addEventListener("gesturechange", handleGestureEvent,
false);
```

最后判定一下浏览器是否支持触屏事件了。

```
    var isIOS = /iP(ad|hone|od)/.test(navigator.userAgent)
    var IE11touch = navigator.pointerEnabled
    var IE9_10touch = navigator.msPointerEnabled
    var w3ctouch = (function() {
        var supported = isIOS || false
        //http://stackoverflow.com/questions/5713393/creating-a
nd-firing-touch-events
        //-on-a-touch-enabled-browser
        try {
            var div = document.createElement("div")
            div.ontouchstart = function() {
                supported = true
            }
            var e = document.createEvent("TouchEvent")
            e.initUIEvent("touchstart", true, true)
            div.dispatchEvent(e)
        } catch (err) {
        }
        div = div.ontouchstart = null
        return supported
    })()
var touchSupported = !!(w3ctouch || IE11touch || IE9_10touch)
```

11.3　tap系事件

tap事件完全是因为那300ms延迟催生出来的产物。PC上点击事件拥有的类型与效果，它也应该全部拥有。比如说，点击事件分单击（click）、双击（dblclick），这个也分单击（tap）、双击（double）。此外，还存在一种叫长按（hold或longtap）的事件。毕竟双击在移动端有特殊任务在身，于是许多双击的任务转交给长按了。

点击事件在上传域上点击，会弹出上传菜单；如果碰到label元素时，会让其for属性对应的目标元素选中；点击提交按钮时，会提交表单。这些行为，使用模拟出来的tap事件是无法实现的，这时，我们需要手动创建一个真正的click或mousedown事件，来实现这些效果。还有要忽略快速触摸而触发的无效点击事件，阻止不想要的放大效果。这一切，在著名的fastclick中都总结好了，因此我们后

人在实现tap事件时，基本遵循它的源码进行改良。

说了这么多，让我们上代码吧。首先要判定当前设备，我们的代码有许多分支是针对不同的版本的iOS或安卓进行打补丁。

```
var ua = navigator.userAgent.toLowerCase()
//http://stackoverflow.com/questions/9038625/detect-if-device-i
s-ios
function iOSversion() {
    if (/iPad|iPhone|iPod/i.test(ua) && !window.MSStream) {
        if ("backdropFilter" in document.documentElement.style)
 {
            return 9
        }
        if (!!window.indexedDB) {
            return 8;
        }
        if (!!window.SpeechSynthesisUtterance) {
            return 7;
        }
        if (!!window.webkitAudioContext) {
            return 6;
        }
        if (!!window.matchMedia) {
            return 5;
        }
        if (!!window.history && 'pushState' in window.history)
{
            return 4;
        }
        return 3;
    }
    return NaN;
}

var deviceIsAndroid = ua.indexOf('android') > 0
var deviceIsIOS = iOSversion()
```

这是来自overflowstack的代码，使用特征侦测来区分iOS的版本。

接下来，我们需要对touchstart、touchmove、touchend、touchcancel进行封装，通过监听整个触摸流程，然后判定其中触摸点的个数，触摸点是否发生移动、移动速度、移动方向及按下时间等，创建出tap、swipe、pinch等一系列自定义事件。

上面的代码及接着下来的代码都是来源于avalon。它们都是基于上一章的avalon事件系统。由于它非常简单于便于移植，大家可以搬到自己的框架中去。

```
var Recognizer = avalon.Recognizer = {
    pointers: {},
    start: function (event, callback) {
        for (var i = 0; i < event.changedTouches.length; i++) {
            var touch = event.changedTouches[i]
            var pointer = {
                startTouch: mixTouchAttr({}, touch),
                startTime: Date.now(),
                status: 'tapping',
```

```
                element: event.target
            }
            Recognizer.pointers[touch.identifier] = pointer;
            callback(pointer, touch)

        }
    },
    move: function (event, callback) {
        for (var i = 0; i < event.changedTouches.length; i++) {
            var touch = event.changedTouches[i]
            var pointer = Recognizer.pointers[touch.identifier]
            if (!pointer) {
                return
            }

            if (!("lastTouch" in pointer)) {
                pointer.lastTouch = pointer.startTouch
                pointer.lastTime = pointer.startTime
                pointer.deltaX = pointer.deltaY = pointer.durat
ion =  pointer.distance = 0
            }

            var time = Date.now() - pointer.lastTime

            if (time > 0) {

                var RECORD_DURATION = 70
                if (time > RECORD_DURATION) {
                    time = RECORD_DURATION
                }
                if (pointer.duration + time > RECORD_DURATION)
{
                    pointer.duration = RECORD_DURATION - time
                }

                pointer.duration += time;
                pointer.lastTouch = mixTouchAttr({}, touch)

                pointer.lastTime = Date.now()

                pointer.deltaX = touch.clientX - pointer.startT
ouch.clientX
                pointer.deltaY = touch.clientY - pointer.startT
ouch.clientY
                var x = pointer.deltaX * pointer.deltaX
```

```
                var y = pointer.deltaY * pointer.deltaY
                pointer.distance = Math.sqrt(x + y)
                pointer.isVertical = !(x > y)

                callback(pointer, touch)
            }
        }
    },
    end: function (event, callback) {
        for (var i = 0; i < event.changedTouches.length; i++) {
            var touch = event.changedTouches[i],
                    id = touch.identifier,
                    pointer = Recognizer.pointers[id]

            if (!pointer)
                continue

            callback(pointer, touch)

            delete Recognizer.pointers[id]
        }
    },
    fire: function (elem, type, props) {
        if (elem) {
            var event = document.createEvent('Events')
            event.initEvent(type, true, true)
            avalon.mix(event, props)
            elem.dispatchEvent(event)
        }
    },
    add: function (name, recognizer) {
        function move(event) {
            recognizer.touchmove(event)
        }

        function end(event) {
            recognizer.touchend(event)

            document.removeEventListener('touchmove', move)
            document.removeEventListener('touchend', end)
            document.removeEventListener('touchcancel', cancel)

        }

        function cancel(event) {
```

```
            recognizer.touchcancel(event)
            document.removeEventListener('touchmove', move)
            document.removeEventListener('touchend', end)
            document.removeEventListener('touchcancel', cancel)

        }

        recognizer.events.forEach(function (eventName) {
            avalon.eventHooks[eventName] = {
                fn: function (el, fn) {
                    if (!el.getAttribute('data-' + name)) {
                        el.setAttribute('data-' + name, '1')
                        el.addEventListener('touchstart', funct
ion (event) {
                            recognizer.touchstart(event)

                            document.addEventListener('touchmov
e', move)
                            document.addEventListener('touchend
', end)
                            document.addEventListener('touchcan
cel', cancel)

                        })
                    }
                    return fn
                }
            }
        })
    }
}


var touchkeys = ['screenX', 'screenY', 'clientX', 'clientY', 'p
ageX', 'pageY']

// 复制 touch 对象上的有用属性到固定对象上
function mixTouchAttr(target, source) {
    if (source) {
        touchkeys.forEach(function (key) {
            target[key] = source[key]
        })
    }
    return target
}
```

从下往上看，我们添加一个系列的自定义事件是通过avalon. Recognizer. add方法实现的。比如说，tap与click是一个系列，doubletap与longtap是一个系列，swipeleft、swiperight、swipeup、swipedown是一个系列……之所以把 tap与doubletap、longtap分开，缘由有二。一是tap多是来自fastclick的源码，而doubletap、longtap则是来自zepto、hammer的实现。二是基于模板化及常用与否，将它们分开可以满足某些挑剔的人群。许多人不想加入不太常用的doubletap、 longtap。在移动端上，jQuery完全是落伍了，其官方出品的jQuery-mobile沦落为二流产品，不过它的一个插件则非常好用：

```
https://github.com/mattbryson/TouchSwipe-Jquery-Plugin
```

add方法的本质是为上一章介绍的eventHooks对象添加钩子函数。钩子函数内部的组织形式其实与实现拖动差不多，都是元素上绑定touchstart事件，当它被触发时，在全局上添加touchmove, touchend, touchcancel事件。其中这4个事件再在回调里面调用各体系添加的touchstart、touchmove、touchend、touchcancel方法。整个add方法的流程如图11-4所示。

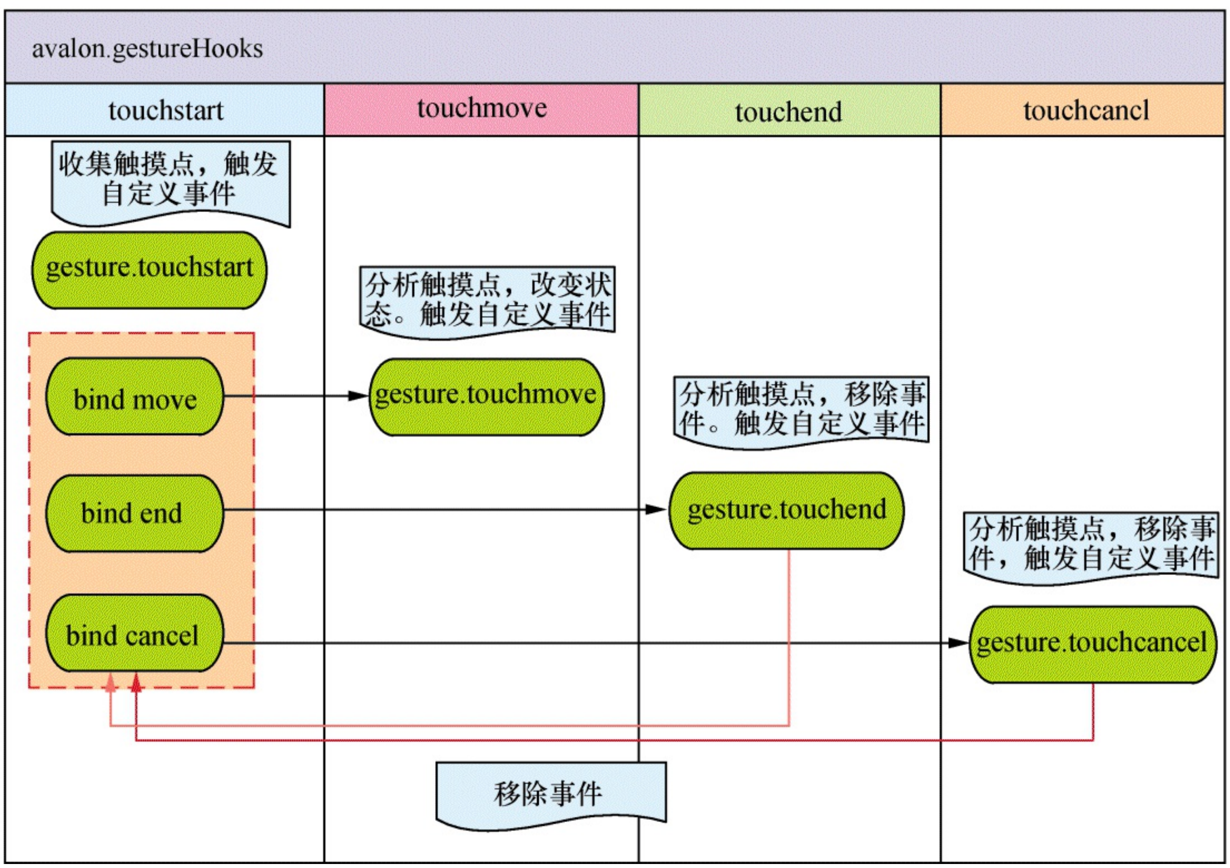


图11-4

接下来我们分析一下Recognizer中其他方法与属性的含义。

pointers是放置触摸点的信息，其键名为Touch对象的identifier属性。它里面包含了触摸的时间（startTime）、位置信息（startTouch，通过mixTouchArr得到）、状态（status）与事件源元素（element）。这些数据在移动过程还会继续添加与修正。

start方法，是用于获取触摸点，及执行对应回调。

move方法，是修正触摸点的信息，如果时间距touchstart时间过去了70ms，那么我们就修正duration、lastTime、distance、lastTouch等属性，并判定其是否在垂直移动。不过垂直与否，其实也不

太准确，只是看它在垂直方向或水平方向哪一个移动距离远一些。执行more方法时会执行相应回调。

end方法，主要是清除触摸点及执行相应回调。

fire方法，是触发自定义事件。这个在上一章已经提到了。这里使用最稳妥的document.createEvent（'Events'）来创建自定义事件。

好了，有了这主体框架，我们就可以通过add方法为它添加tap、swipe等事件了。我们可以将tap相关的内容独立成一个JS文件，如touch.tap.js，里面组织模块的形式可以选用AMD、CMD、CommonJS……

```
var supportPointer = !!navigator.pointerEnabled || !!navigator.
msPointerEnabled
//root为document.documentElement
if (supportPointer) { // 支持pointer的设备可用样式来取消click事件的3
00毫秒延迟
    root.style.msTouchAction = root.style.touchAction = 'none'
}
var tapRecognizer = {
```

```
    events: ['tap', 'click'],
    touchBoundary: 10,
    tapDelay: 200,
    /**
    needClick(),
    needFocus(),
    fixTarget()
    focus(),
    updateScrollParent()
    touchHasMoved(),
    findControl(),
    findType(),
    sendClick(),
    */
    touchstart: function (event) {},
    touchmove: function (event) {},
    touchend: function (event) {},
    touchcancel: function () {}
}
Recognizer.add("tap", tapRecognizer)
```

如果读过fastclick的读者，就会发现其中needClick、needFocus、findControl、sendClick与fastclick的内部方法很像。不错，本模块是大量参考fastclick编写的。touchBoundary表示某一触摸点的移动距离，如果它在某一方向移动超过10px，我们就将它当成其他事件了。tapDelay表示两次点击事件应该相隔多长时间，短于这个我们就忽略掉后面那个。

我们还是顺着touchstart、touchmove、touchend、touchend的顺序看起，如果其中涉及其他方法，再读那些方法。

```
touchstart: function (event) {
    //忽略多点触摸
    if (event.targetTouches.length !== 1) {
        return true
    }
    //修正事件源对象
    var targetElement = tapRecognizer.fixTarget(event.target)
    var touch = event.targetTouches[0]
    if (deviceIsIOS) {
        // 判断是否是点击文字，进行选择等操作，如果是，不需要模拟click
        var selection = window.getSelection();
        if (selection.rangeCount && !selection.isCollapsed) {
            return true
        }
        var id = touch.identifier
        //当 alert 或 confirm 时，点击其他地方，会触发touch事件，iden
tifier相同，此事件应该被忽略
        if (id && isFinite(tapRecognizer.lastTouchIdentifier) &
&
        tapRecognizer.lastTouchIdentifier === id) {
            event.preventDefault()
            return false
        }

        tapRecognizer.lastTouchIdentifier = id

        tapRecognizer.updateScrollParent(targetElement)
    }
    //收集触摸点的信息
    tapRecognizer.status = "tapping"
    tapRecognizer.startTime = Date.now()
    tapRecognizer.element = targetElement
    tapRecognizer.pageX = touch.pageX
    tapRecognizer.pageY = touch.pageY
    // 如果点击太快,阻止双击带来的放大收缩行为
    if ((tapRecognizer.startTime - tapRecognizer.lastTime) < ta
```

```
pRecognizer.tapDelay) {
        event.preventDefault()
    }
},
```

touchstart主要是收集情报。由于是点击事件，因此我们只处理一个触摸点的情况，发现有两个或两个以上的指头在屏幕上，应该立即返回。然后找到正确的事件源对象，再然后处理iOS上的各种怪事，然后往上看一下其祖先元素是否存在滚动条，保持此祖先及滚动位置，方便以后作比较。最后是收集触摸点的相关信息与阻止双击。这里面用到两个私有方法fixTarget与 updateScrollParent。

```
 fixTarget: function (target) {
     if (target.nodeType === 3) {
         return target.parentNode
     }
     if (window.SVGElementInstance && (target instanceof SVGEle
mentInstance)) {
         return target.correspondingUseElement;
     }

     return target
 },
 updateScrollParent: function (targetElement) {
     //如果事件源元素位于某一个有滚动条的祖父元素中,那么保持其scrollPare
nt与scrollTop值
     var scrollParent = targetElement.tapScrollParent
```

```
    if (!scrollParent || !scrollParent.contains(targetElement)
) {
        var parentElement = targetElement;
        do {
            if (parentElement.scrollHeight > parentElement.off
setHeight) {
                scrollParent = parentElement;
                targetElement.tapScrollParent = parentElement
                break;
            }
            parentElement = parentElement.parentElement
        } while (parentElement);
    }
    if (scrollParent) {
        scrollParent.lastScrollTop = scrollParent.scrollTop
    }
 },
```

touchmove的目标很单一，判定其是发生移动，移动了就改变其状态，清空事件源元素。

```
touchmove: function (event) {
    if (tapRecognizer.status !== "tapping") {
        return true
    }
    // 如果事件源元素发生改变,或者发生了移动,那么就取消触发点击事件
    if (tapRecognizer.element !== tapRecognizer.fixTarget(event
.target) ||
            tapRecognizer.touchHasMoved(event)) {
        tapRecognizer.status = tapRecognizer.element = 0
    }
},
touchHasMoved: function (event) {
    // 判定是否发生移动,其阈值是10px
    var touch = event.changedTouches[0],
             boundary = tapRecognizer.touchBoundary
   return Math.abs(touch.pageX - tapRecognizer.pageX) > boundar
y ||
```

```
        Math.abs(touch.pageY - tapRecognizer.pageY) > boundary
},
```

其中needFocus、needClick、focus、findControl方法是精华，是人们经过千万次调试总结出来的。

```
needClick: function (target) {
    //判定是否使用原生的点击事件, 否则使用sendClick方法手动触发一个人工的点击事件
    switch (target.nodeName.toLowerCase()) {
        case 'button':
        case 'select':
        case 'textarea':
            if (target.disabled) {
                return true
            }

            break;
        case 'input':
        // IOS6 pad 上选择文件，如果不是原生的click，弹出的选择界面尺寸错误
            if ((deviceIsIOS && target.type === 'file') || target.disabled) {
                return true
            }

            break;
        case 'label':
        case 'iframe':
        case 'video':
            return true
    }

    return false
},
needFocus: function (target) {
    switch (target.nodeName.toLowerCase()) {
        case 'textarea':
```

```
        case 'select': //实测android下select也需要
            return true;
        case 'input':
            switch (target.type) {
                case 'button':
                case 'checkbox':
                case 'file':
                case 'image':
                case 'radio':
                case 'submit':
                    return false
            }
            //如果是只读或disabled状态,就无须获得焦点了
            return !target.disabled && !target.readOnly
        default:
            return false
    }
},
focus: function (targetElement) {
    var length;
    //在iOS7下, 对一些新表单元素(如date、datetime、time、month)调用fo
cus方法会抛错,
    //幸好的是,我们可以改用setSelectionRange获取焦点, 将光标挪到文字的
最后
    var type = targetElement.type
    if (deviceIsIOS && targetElement.setSelectionRange &&
            type.indexOf('date') !== 0 && type !== 'time' && ty
pe !== 'month') {
        length = targetElement.value.length
        targetElement.setSelectionRange(length, length)
    } else {
        targetElement.focus()
    }
},
findControl: function (labelElement) {
    // 获取label元素所对应的表单元素
    // 可以能过control属性, getElementById, 或用querySelector直接找
其内部第一表单元素实现
    if (labelElement.control !== undefined) {
        return labelElement.control
    }

    if (labelElement.htmlFor) {
        return document.getElementById(labelElement.htmlFor)
    }
```

```
    return labelElement.querySelector('button, input:not([type=hidden]), keygen, meter, output, progress, select, textarea')
},
```

此外还有一个sendClick方法，有些时候我们不得不使用原生的点击事件，比如我们点击了submit按钮要提交表单，这是自定义事件所无能为力的。不过有些时候，光是click事件又不顶用，需要用到mousedown事件，于是我们又分化出findType这个辅助方法。

```
findType: function (targetElement) {
    // 安卓chrome浏览器上，模拟的 click 事件不能让 select 打开，故使用 mousedown 事件
    return deviceIsAndroid && targetElement.tagName.toLowerCase() === 'select' ?
            'mousedown' : 'click'
},
sendClick: function (targetElement, event) {
    // 在click之前触发tap事件
    Recognizer.fire(targetElement, 'tap', {
        fastclick: true
    })
    var clickEvent, touch
    //某些安卓设备必须先移除焦点，之后模拟的click事件才能让新元素获取焦点
    if (document.activeElement && document.activeElement !== targetElement) {
        document.activeElement.blur()
    }

    touch = event.changedTouches[0]
```

```
    // 手动触发点击事件,此时必须使用document.createEvent('MouseEvents')来创建事件
    // 及使用initMouseEvent来初始化它
    clickEvent = document.createEvent('MouseEvents')
    clickEvent.initMouseEvent(tapRecognizer.findType(targetElement), true, true, window,
    1, touch.screenX,
    touch.screenY, touch.clientX, touch.clientY, false, false, false, false, 0, null)
    clickEvent.fastclick = true
    targetElement.dispatchEvent(clickEvent)
},
```

最后是touchcancel方法，就是做一些清空工作。

```
touchcancel: function () {
   tapRecognizer.startTime = tapRecognizer.element = 0
}
```

11.4　press系事件

tap系事件可谓最复杂，为了解决300ms延迟而伪造的点击事件，需要摸拟一切与真实click事件相关的行为。但这一步迈过去，剩下的路就好走了。现在我们看双击与长按，如图11-5所示。

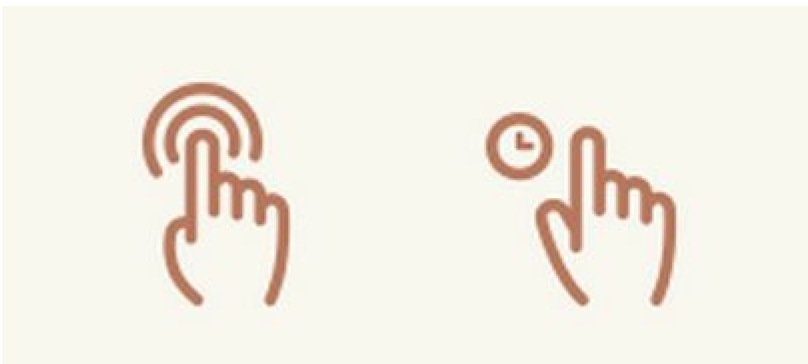


图11-5

我们可以使用一个touch.press.js文件独立放置它的代码，由于大量功能使用了Recognizer.start、Recognizer.move、Recognizer.end方法进行复用了，代码量比较少，我们就一气呵成，全部贴上来吧。

```
var pressRecognizer = {
```

```
    events: ['longtap', 'doubletap'],
    cancelPress: function (pointer) {
        clearTimeout(pointer.pressingHandler)
        pointer.pressingHandler = null
    },
    touchstart: function (event) {
        Recognizer.start(event, function (pointer, touch) {
            pointer.pressingHandler = setTimeout(function () {
                if (pointer.status === 'tapping') {
                    Recognizer.fire(event.target, 'longtap', {
                        touch: touch, touchEvent: event
                    })
                }
                pressRecognizer.cancelPress(pointer)
            }, 500)
            if (event.changedTouches.length !== 1) {
                pointer.status = 0
            }
        })

    },
    touchmove: function (event) {
        Recognizer.move(event, function (pointer) {
            if (pointer.distance > 10 && pointer.pressingHandle
r) {
                pressRecognizer.cancelPress(pointer)
                if (pointer.status === 'tapping') {
                    pointer.status = 'panning'
                }
            }
        })
    },
    touchend: function (event) {
        Recognizer.end(event, function (pointer, touch) {
            pressRecognizer.cancelPress(pointer)
if (pointer.status === 'tapping') {
    pointer.lastTime = Date.now()
                if (pressRecognizer.lastTap && pointer.lastTime
 - pressRecognizer.
                lastTap.lastTime < 300) {
                    Recognizer.fire(pointer.element, 'doubletap
', {
                        touch: touch,
                        touchEvent: event
                    })
```

```
                }

                pressRecognizer.lastTap = pointer
            }
        })

    },
    touchcancel: function (event) {
        Recognizer.end(event, function (pointer) {
            pressRecognizer.cancelPress(pointer)
        })
    }
}
Recognizer.add('press', pressRecognizer)
```

首先看长按是怎么实现，在touchstart中，我们默认每个触摸点（pointer）的状态为tapping，如果同时存在两个触摸点，我们就置为0。此时我们为触摸点添加一个setTimeout回调，过期时间为500ms，有的库也使用750ms，在它们之间为妥。只要在这时间它没有clearTimeout，就会自动触发长按事件。touchmove回调就是检测触摸点是否发生位移，发生了就clearTimeout，并改变状态。在touchend中，我们也会clearTimeout，说明这个触摸行为过短了。此外，还判定两次触摸行为是否少于

300ms，是就触发双击事件。

如果大家会avalon，可以使用以下页面进行测试。

```
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>touch</title>
        <script src="dist/avalon.mobile.js"></script>
        <meta id="viewport" name="viewport" content="initial-sc
ale=1.0,user-scalable=no,
              minimum-scale=1.0, maximum-scale=1.0">
        <style>
            .longtap{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background: rosybrown
            }
            .doubletap{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:gold;
            }

        </style>
    </head>
    <body ms-controller="test">

        <div class="longtap" ms-on-longtap="press">

        </div>
        <div class="doubletap" ms-on-doubletap="press">
            //双击
        </div>
```

```
        <script type="text/javascript">

            var vm = avalon.define({
                $id: "test",
                press: function (e) {
                    console.log(e.type)
                    this.innerHTML = e.type + " " + (new Date -
 0)
                }
            })

        </script>
    </body>
</html>
```

注意，在长按过程中，可能触发系统右键菜单，可以使用以下方式禁用掉。

```
element.addEventListener("MSHoldVisual", function(e) { e.preven
tDefault() }, false)
element.addEventListener("contextmenu", function(e) { e.prevent
Default() }, false)
```

11.5　swipe系事件

swipe意即划动，要用速度感，因此其限制条件之一是速度，主流的限制速度是0.65px/ms。此外，划动只要一个手指头就够了，还有划动要有距离。而这所有要素，我们已经在Recognizer. start、Recognizer. move、Recognizer. end方法进行复用了，因此其代码量也比较少。最难的是方向的判定，要用到一些数学知识，如图11-6所示。

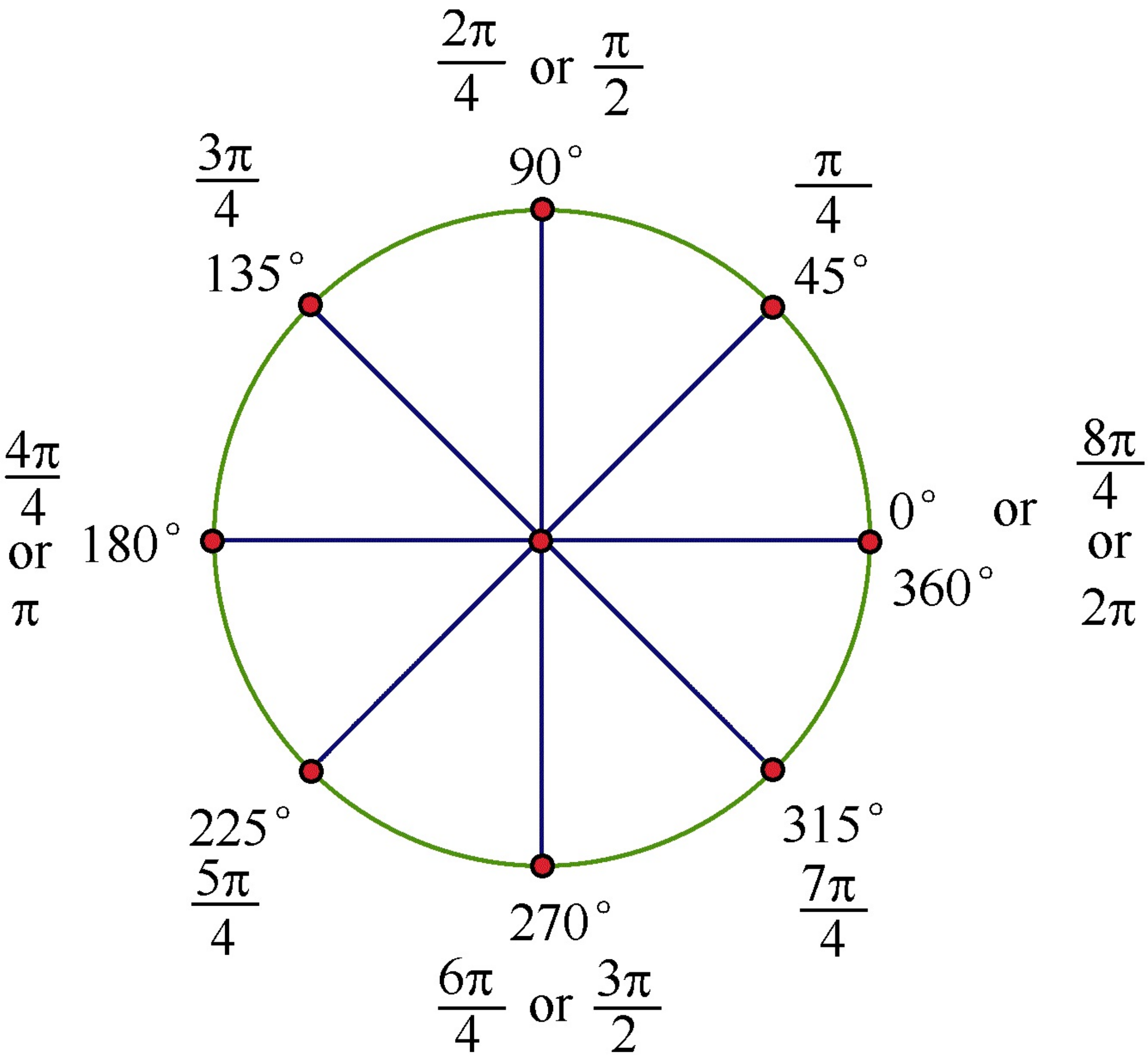


图11-6

我们可以使用一个touch.press.js文件独立放置它的代码。其主体结构如下：

```
var swipeRecognizer = {
    events: ['swipe', 'swipeleft', 'swiperight', 'swipeup', 'sw
ipedown'],
    getAngle: function (x, y) {
```

```
        var r = Math.atan2(y, x) //radians
        var angle = Math.round(r * 180 / Math.PI) //degrees
       },
    getDirection: function (startPoint, endPoint) {
        var angle = swipeRecognizer.getAngle(startPoint, endPoi
nt)
        if ((angle < -45) && (angle > -135)) {
            return "up"
        } else if ((angle >= 45) && (angle < 315)) {
            return "down"
        } else if ((angle > -45) && (angle <= 45)) {
            return "right"
        } else{
            return "left"
        }
    },
    touchstart: function (event) {
    },
    touchmove: function (event) {
    },
    touchend: function (event) {
    }
}

swipeRecognizer.touchcancel = swipeRecognizer.touchend
Recognizer.add('swipe', swipeRecognizer)
```

如果大家会avalon，可以使用以下页面进行测试。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>touch</title>
        <script src="../../dist/avalon.mobile.js"></script>
        <meta id="viewport" name="viewport" content="initial-sc
ale=1.0,user-scalable=no,
```

```
            minimum-scale=1.0, maximum-scale=1.0">
        <style>
            .swipeleft{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:aqua;
            }
            .swiperight{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:khaki;
            }
            .swipeup{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:blueviolet;
            }
            .swipedown{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:lawngreen;
            }

        </style>
    </head>
    <body ms-controller="test">
        <h1>swipe</h1>
        <div class="swipeleft" ms-on-swipeleft="swipe">
            left
        </div>
        <div class="swiperight" ms-on-swiperight="swipe">
            right
        </div>
        <div class="swipeup" ms-on-swipeup="swipe">
            up
        </div>
        <div class="swipedown" ms-on-swipedown="swipe">
```

```
            down
        </div>
        <script type="text/javascript">

            var vm = avalon.define({
                $id: "test",
                swipe: function (e) {
                    console.log(e.type)
                    this.innerHTML = e.type + " " + (new Date - 
 0)
                }
            })

        </script>
    </body>
</html>
```

11.6　pinch系事件

这个事件也叫缩放（scale），主要涉及两个行为pinchin（缩小，向里捏合，两只手指的距离越来越近）和pinchout（也叫spread，放大，向外扩张，两只手指的距离越来越远）。从描述来看，就涉及两个触摸点。此外，为了方便我们在开始或结束时处理些事情，我们通常把这家族扩展为5个事件：pininstart、pinch、pinchin、pinchout、pinchend。此类事件经常用于查看图片，如图11-7所示。我们可以使用一个touch.pinch.js文件独立放置它的代码。其主体结构如下：

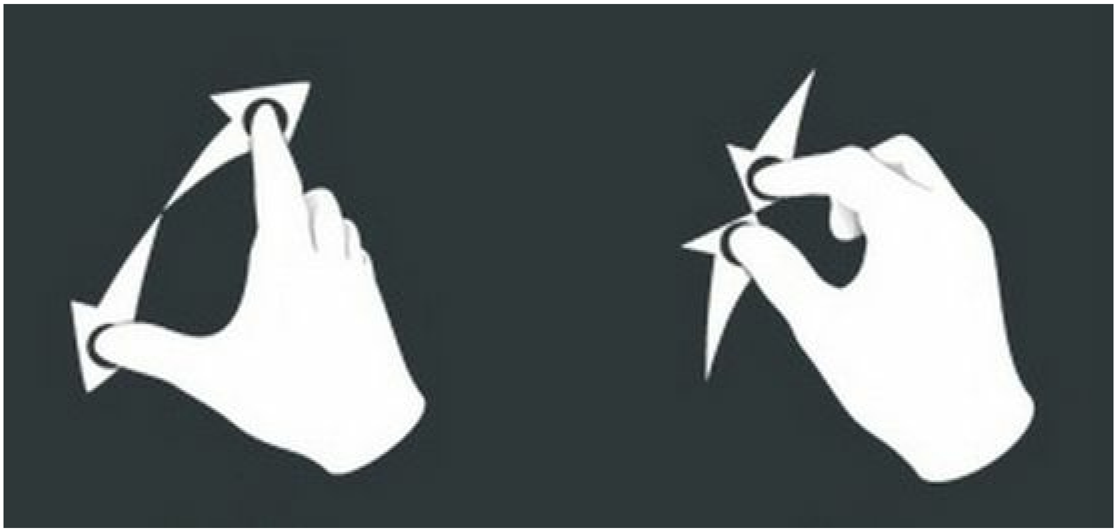


图11-7

```
var pinchRecognizer = {
    events: ['pinchstart', 'pinch', 'pinchin', 'pinchuot', 'pin
chend'],
    getScale: function (x1, y1, x2, y2, x3, y3, x4, y4) {
        return Math.sqrt((Math.pow(y4 - y3, 2) + Math.pow(x4 -
x3, 2)) / (Math.pow(y2 –
        y1, 2) + Math.pow(x2 - x1, 2)))
    },
    getCommonAncestor: function (arr) {
        var el = arr[0], el2 = arr[1]
        while (el) {
            if (el.contains(el2) || el === el2) {
                return el
            }
            el = el.parentNode
        }
        return null
    },
    touchstart: function (event) {},
    touchmove: function (event) {},
    touchend: function (event)  {}
}

pinchRecognizer.touchcancel = pinchRecognizer.touchend

Recognizer.add('pinch', pinchRecognizer)
```

这里涉及两个辅助方法，第一个是求放缩比，路思是计算前后这两点的距离的比率，第二个是求两个触摸点的最近公共祖先。

```
 touchstart: function (event) {
    var pointers = Recognizer.pointers
    Recognizer.start(event, avalon.noop)
    var elements = []
    for (var p in pointers) {
        if (pointers[p].startTime) {
            elements.push(pointers[p].element)
        } else {
            delete pointers[p]
        }
    }
    pointers.elements = elements
    if (elements.length === 2) {
        pinchRecognizer.element = pinchRecognizer.getCommonAnce
stor(elements)
        Recognizer.fire(pinchRecognizer.getCommonAncestor(eleme
nts), 'pinchstart', {
            scale: 1,
            touches: event.touches,
            touchEvent: event
        })
    }
},
```

接下来的touchstart回调中，我们需要触发pinchstart方法。我们在公共的Recognizer.start方法

中已经将所有触摸点储存在Recognizer.pointers对象上，这时我们将它转换成一个数组，判定个数是否为2。默认的放缩比scale为1。

```
touchmove: function (event) {
    if (pinchRecognizer.element && event.touches.length > 1) {
        var position = [],
                current = []
        for (var i = 0; i < event.touches.length; i++) {
            var touch = event.touches[i];
            var gesture = Recognizer.pointers[touch.identifier]
;
            position.push([gesture.startTouch.clientX, gesture.
startTouch.clientY]);
            current.push([touch.clientX, touch.clientY]);
        }

        var scale = pinchRecognizer.getScale(position[0][0], po
sition[0][1], position[1][0], position[1][1],  current[0][0], c
urrent[0][1], current[1][0], current[1][1]);
        pinchRecognizer.scale = scale
        Recognizer.fire(pinchRecognizer.element, 'pinch', {
            scale: scale,
            touches: event.touches,
            touchEvent: event
        })

        if (scale > 1) {
            Recognizer.fire(pinchRecognizer.element, 'pinchout'
, {
                scale: scale,
                touches: event.touches,
                touchEvent: event
            })

        } else {
            Recognizer.fire(pinchRecognizer.element, 'pinchin',
 {
                scale: scale,
                touches: event.touches,
```

```
                touchEvent: event
            })
        }
     }
     event.preventDefault()
  },
```

touchmove回调比较复杂，我们需要在里面区分放大或是缩小。我们需要取得当前两个指头的坐标，也就是pageX、pageY。然后一共4个点、8个参数放进getScale方法里计算放缩比。然后根据放缩比，选择触发pinchin、pinchout事件。为了方便一些懒人，我们也提供了pinch事件。最后的**event.preventDefault()** 是为了阻止文本被选择。

```
touchend: function (event) {
    if (pinchRecognizer.element) {
        Recognizer.fire(pinchRecognizer.element, 'pinchend', {
            scale: pinchRecognizer.scale,
            touches: event.touches,
            touchEvent: event
        })
        pinchRecognizer.element = null
    }
    Recognizer.end(event, avalon.noop)
}
```

touchend回调则比较简单，放缩比是上一个touchmove回调中保留下来的，直接用到pinchend事件上。最后做一些清除工作。

如果大家会avalon，可以使用以下页面进行测试。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>touch</title>
        <script src="../../dist/avalon.mobile.js"></script>
        <meta id="viewport" name="viewport" content="initial-sc
ale=1.0,user-scalable=no,
              minimum-scale=1.0, maximum-scale=1.0">
        <style>
            .pinch{
                width:230px;
                height:230px;
                overflow: hidden;
                display: inline-block;
                background:red;
            }
        </style>
    </head>
    <body ms-controller="test">
        <h1>pinch, pinchstart, pinchend, pinchin, pinchout</h1>
        <div class="pinch"
             ms-on-pinchstart='pinch'
             ms-on-pinchend='pinch'
             ms-on-pinch="pinch"
             ms-on-pinchin="scale"
             ms-on-pinchout="scale">
             <p>{{a}}</p>
             <p>{{b}}</p>
```

```
        </div>
        <script type="text/javascript">

    var vm = avalon.define({
        $id: "test",
        a: "",
        b: "",
        pinch: function (e) {
            vm.a = e.type
        },
        scale: function (e) {
            vm.b = e.type + " " + e.scale
            if ("transform" in this.style) {
                this.style.transform = 'scale(' + e.scale + ')'
;
            } else {
                this.style.webkitTransform = 'scale(' + e.scale
 + ')';
            }

        }
    })
        </script>
    </body>
</html>
```

11.7　拖放系事件

这个其实与PC端的实现原理差不多，在鼠标（手指）放下时，收集坐标，移动时，得到改变距离，加上最开始的top、left样式值上，就能拖动目标元素了。不过在移动端上，我们还可以使用transform样式实现。此类事件应该是最常用的事件了，各种弹出层的拖曳，菜单的排序，游戏元素的移动……如图11-8所示。

角度的计算，如图11-9所示。

我们可以使用一个touch.drag.js文件独立放置它的代码。其代码如下。



图11-8

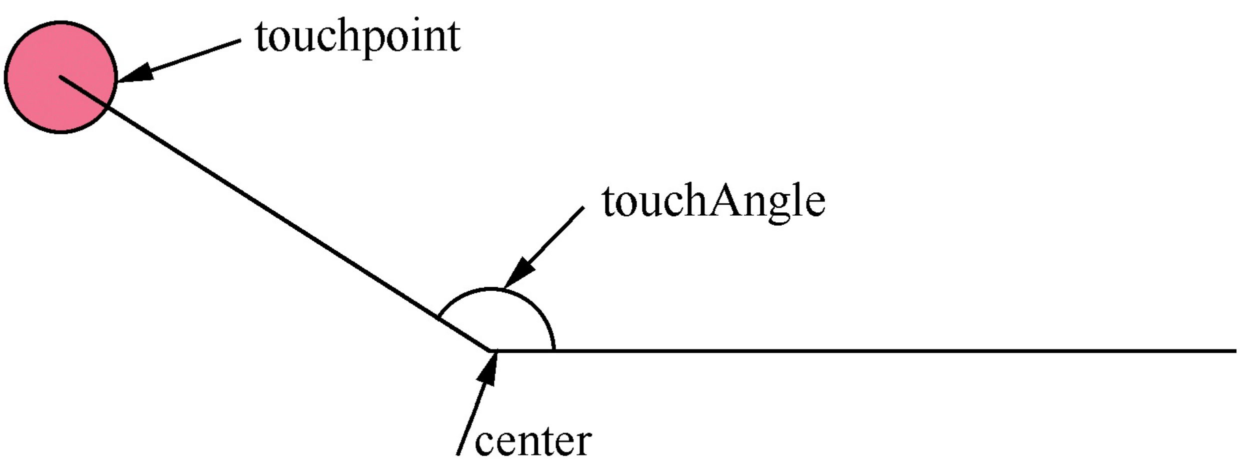


图11-9

```
var Recognizer = avalon.gestureHooks
var dragRecognizer = {
    events: ['dragstart', 'drag', 'dragend'],
    touchstart: function (event) {
        Recognizer.start(event, avalon.noop)
    },
```

```
    touchmove: function (event) {
        Recognizer.move(event, function (pointer, touch) {
            var extra = {
                deltaX: pointer.deltaX,
                deltaY: pointer.deltaY,
                touch: touch,
                touchEvent: event,
                isVertical: pointer.isVertical
            }
            if ((pointer.status === 'tapping') && pointer.dista
nce > 10) {
                pointer.status = 'panning'
                Recognizer.fire(pointer.element, 'dragstart', e
xtra)
            } else if (pointer.status === 'panning') {
                Recognizer.fire(pointer.element, 'drag', extra)
            }
        })

        event.preventDefault();
    },
    touchend: function (event) {
        Recognizer.end(event, function (pointer, touch) {
            if (pointer.status === 'panning') {
                Recognizer.fire(pointer.element, 'dragend', {
                    deltaX: pointer.deltaX,
                    deltaY: pointer.deltaY,
                    touch: touch,
                    touchEvent: event,
                    isVertical: pointer.isVertical
                })
            }
        })
        Recognizer.pointers = {}
    }
}
dragRecognizer.touchcancel = dragRecognizer.touchend

Recognizer.add('drag', dragRecognizer)
```

大体结构与swipe差不多，许多工作交由

Recognizer.start、Recognizer. move、Recognizer.end 复用了。

如果大家会avalon，可以使用以下页面进行测试。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>touch</title>
        <script src="../../dist/avalon.mobile.js"></script>
        <meta id="viewport" name="viewport" content="initial-sc
ale=1.0,user-scalable=no,
              minimum-scale=1.0, maximum-scale=1.0">
        <style>
            .drag{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:aqua;
            }
        </style>
    </head>
    <body ms-controller="test">
        <h1>drag,dragstart,dragend</h1>
        <div class="drag"
             ms-on-dragstart="dragstart"
             ms-on-drag="drag"
             ms-on-dragend="dragend">
            <p>{{a}}</p>
            <p>{{b}}</p>
            <p>{{c}}</p>
        </div>

        <script type="text/javascript">
    var dx = 0, dy = 0;
```

```
    var vm = avalon.define({
        $id: "test",
        a: "",
        b: "",
        c: "",
        dragstart: function (e) {
            vm.a = e.type
            vm.c = " "
        },
        drag: function (e) {
            var x = dx + e.deltaX
            var y = dy + e.deltaY
            var offx = x + "px";
            var offy = y + "px";
            vm.b = offx + " drag " + offy
            if ("transform" in this.style) {
                this.style.transform = "translate3d(" + offx +
"," + offy + ",0)";
            } else {
                this.style.webkitTransform = "translate3d(" + o
ffx + "," + offy + ",0)";
            }
        },
        dragend: function (e) {
            vm.a = " "
            dx = e.touch.pageX
            dy = e.touch.pageY
            vm.c = e.type
        }
    })

        </script>
    </body>
</html>
```

11.8　rotate系事件

旋转在移动端也非常常用，它是以手指所在的元素的中心为圆心，手指在它的某一边缘，然后拖动它实现旋转。它可以说是拖拽事件的变种。在本书中的旋转事件是单指头的，在网上也流行另一种实现，以hammer.js为首，双指头实现旋转。

宣传活动页面经常见的抽奖转盘（见图11-10）。



图11-10

我们可以使用一个touch.rotate.js文件独立放置它的代码。其代码如下。

```
var rotateRecognizer = {
    events: ['rotate', 'rotatestart', 'rotateend'],
```

```
    getAngle180: function (p1, p2) {
        // 角度， 范围在{0°～180°}， 用来识别旋转角度
        var agl = Math.atan((p2.pageY - p1.pageY) * -1 / (p2.pa
geX - p1.pageX)) * (180 / Math.PI)
        return parseInt((agl < 0 ? (agl + 180) : agl), 10)
    },
    rotate: function (event, status) {
    },
    touchstart: function (event) {
        var pointers = Recognizer.pointers
        Recognizer.start(event, avalon.noop)
        var finger
        for (var p in pointers) {
            if (pointers[p].startTime) {
                if (!finger) {
                    finger = pointers[p]
                } else {//如果超过一个指头就中止旋转
                    return
                }
            }
        }
        rotateRecognizer.finger = finger
        var el = finger.element
        var docOff = avalon(el).offset()
        rotateRecognizer.center = {//求得元素的中心
            pageX: docOff.left + el.offsetWidth / 2,
            pageY: docOff.top + el.offsetHeight / 2
        }
        rotateRecognizer.startAngel = rotateRecognizer.getAngle
180(rotateRecognizer.center, finger.startTouch)
    },
    touchmove: function (event) {
        Recognizer.move(event, avalon.noop)
        rotateRecognizer.rotate(event)
    },
    touchend: function (event) {
        rotateRecognizer.rotate(event, "end")
        Recognizer.end(event, avalon.noop)
    }
}

rotateRecognizer.touchcancel = rotateRecognizer.touchend

Recognizer.add('rotate', rotateRecognizer)
```

这里单独将rotate方法挑出来详细讲。我们先看一下touchstart回调，它的作用是判定当前是否只有一个触摸点，并求出元素的中心，这时就可以得到其初始角度。

```
rotate: function (event, status) {
    var finger = rotateRecognizer.finger
    var endAngel = rotateRecognizer.getAngle180(rotateRecognize
r.center, finger.lastTouch)
    var diff = rotateRecognizer.startAngel - endAngel
    var direction = (diff > 0 ? 'right' : 'left')
    var count = 0;
    var __rotation = ~~finger.element.__rotation
    while (Math.abs(diff - __rotation) > 90 && count++ < 50) {
        if (__rotation < 0) {
            diff -= 180
        } else {
            diff += 180
        }
    }
    var rotation = finger.element.__rotation = __rotation = dif
f
    rotateRecognizer.endAngel = endAngel
    var extra = {
        touch: event.changedTouches[0],
        touchEvent: event,
        rotation: rotation,
        direction: direction
    }
    if (status === "end") {
        Recognizer.fire(finger.element, 'rotateend', extra)
        finger.element.__rotation = 0
    } else if (finger.status === 'tapping' && diff) {
        finger.status = "panning"
        Recognizer.fire(finger.element, 'rotatestart', extra)
```

```
    } else {
        Recognizer.fire(finger.element, 'rotate', extra)
    }
},
```

rotate需要计算当前是往哪个方向旋转，或者说是顺时针还是逆时针。此外还有一个循环，将太大的角度进行收缩，比如370°相当于10°。最后将这角度放到元素的一个属性上。这个属性在移动时一定被重写，直到旋转结束。

如果大家会avalon，可以使用以下页面进行测试。

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>touch</title>
        <script src="../../dist/avalon.mobile.js"></script>
        <meta id="viewport" name="viewport" content="initial-sc
ale=1.0,user-scalable=no,
              minimum-scale=1.0, maximum-scale=1.0">
        <style>
            .rotate{
                width:200px;
                height:200px;
                overflow: hidden;
                display: inline-block;
                background:aqua;
```

```
                border-radius: 100px;
            }

        </style>
    </head>
    <body ms-controller="test">
        <h1>drag,dragstart,dragend</h1>
        <div class="rotate"
             ms-on-rotate="rotate"
             ms-on-rotatestart="rotatestart"
             ms-on-rotateend="rotateend"
             >
            <p>000000</p>
            <p>111111</p>
            <p>222222 {{a}}</p>
            <p>333333 {{b}}</p>
            <p>{{c}}</p>
        </div>

        <script type="text/javascript">

    var vm = avalon.define({
        $id: "test",
        a: "",
        b: "",
        c: "",
        rotatestart: function (e) {
            vm.a = e.type
            vm.b = vm.c = ""
        },
        rotateend: function (e) {
            vm.a = e.type
            vm.b = e.direction
        },
        rotate: function (e) {
            var a = e.rotation
            vm.c = "旋转了" + a + " 度"
            this.style.webkitTransform = 'rotate(' + a + 'deg)'
;
        }
    })

        </script>
    </body>
</html>
```

11.9　总结

由于苹果早期的设计失误及人们对交互要求的提高，因此如你们所见，这些事件都是合成事件。它们都是根据那4、5个事件开发出来的，在某一个阶段得到触摸点的个数，触摸轨迹，识别为某一个自定义事件。因此大家如果想开发自己的手势库，必须有识别器的意识，这样就能节省大量的代码。此外，本章没有提及如何兼容微软平台的事件，不过本人认为微软提供的原生事件比touch系事件好用得多，大家也可以模拟微软的思路开发手势库。

最后奉上一些链接让大家参考一下：

```
http://blogs.msdn.com/b/ie/archive/2012/06/20/go-beyond-pan-zoom-and-tap-using-gesture-events.aspx https://msdn.microsoft.com/en-us/library/windows/apps/hh441180.aspx
```

严格来说，手机上的事件分三大类。

（1）触屏事件：通过触摸，手势进行触发（如手指点击，缩放）。

（2）运动事件：通过加速器进行触发（如手机晃动）。

（3）远程控制事件：通过其他远程设备触发（如耳机控制按钮）但除了触屏事件，其他两个，设备不支持就无法通过单纯的JS手段摸拟了。因此本章就只介绍触屏事件。

第12章　异步模型



浏览器环境与后端的Node.js存在着各种消耗巨大或堵塞线程的行为，对于JavaScript这样单线程的东西唯一的解耦方法就是提供异步API。异步API是怎么样的呢？简单来说，它是不会立即执行的方

法。比方说，一个长度为1000的数组，在for循环内，可能不到几毫秒就执行完毕，若在后端的其他语言，则耗时更少。但有时候，我们不需要这么快的操作，我们想在页面上能用肉眼看到它执行的每一步，那就需要异步API。还有些操作，比如加载资源，你想快也快不了，它不可能一下子提供给你，你必须等待，但你也不能一直干等下去什么也不干，得允许我们跳过这些加载资源的逻辑，执行下面的代码。于是浏览器首先搞出的两个异步API，就是setTimeout与setInterval。后面开始出现各种事件回调，它只有用户执行了某种操作后才触发。再之后，就更多，XMLHttpRequest、postMessage、WebWorkor、setImmediate、requestAnimation Frame等（见图12-1）。

这些东西都有一个共同的特点，就是拥有一个回调函数，描述一会儿要干什么。有的异步API还提供了对应的中断API，比如clearTimeout、

clearInterval、clearImmediate、cancelAnimationFrame。

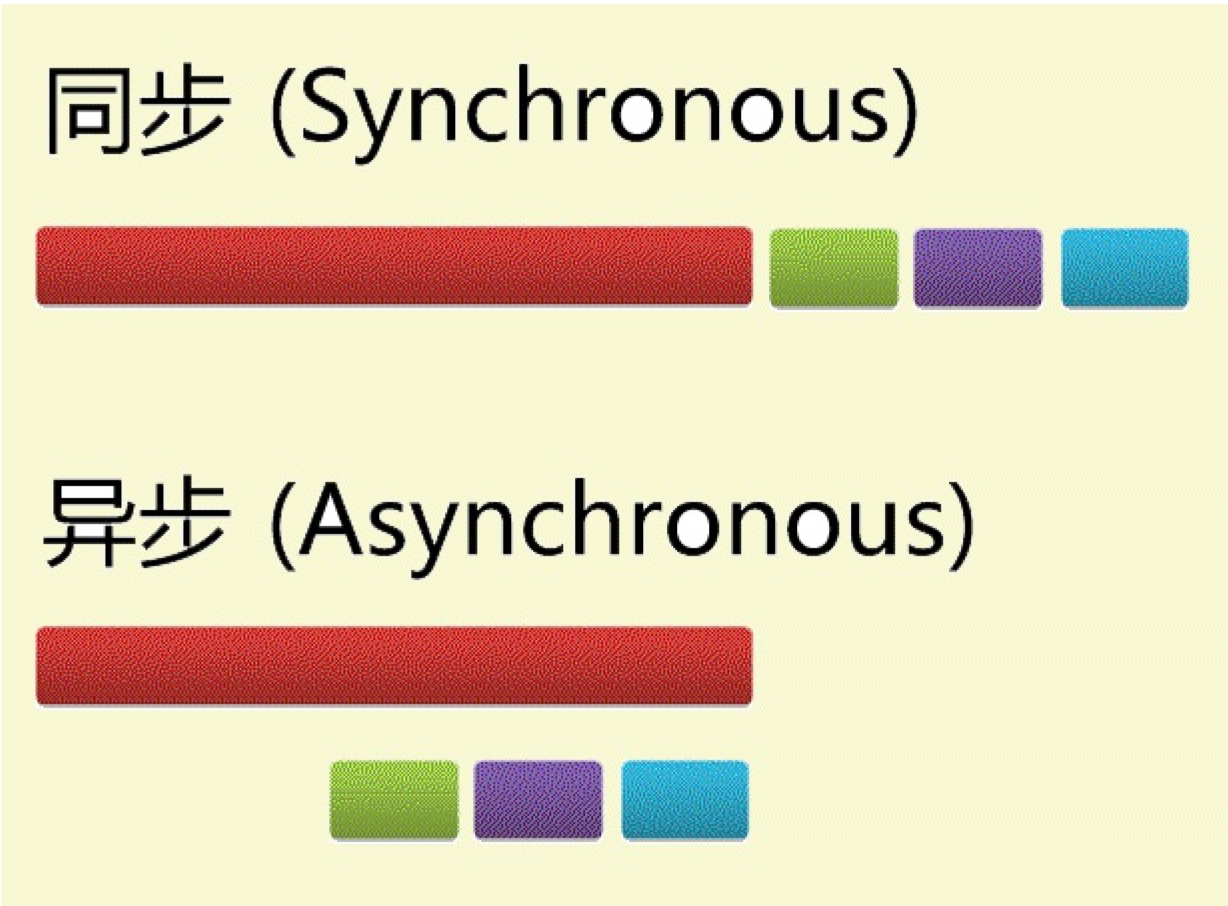


图12-1

早些年，我们就是通过setIimeout或setInterval在网页上实现动画的。这种动画其实就是通过这些异步API不断反复调用同一个回调实现的，回调里

面是对元素节点的某些样式进行很小范围的改动。

随着iframe的挖掘与XMLHttpRequest的出现，无缝刷新让用户驻留在同一个页面上的时间越来越长，许多功能都集成在同一个页面。为实现这些功能，我们就得从后端加载数据与模板，来拼装这些新区域。这些加载数据与模板的请求可能是并行的，可能是存在依赖的。只有在所有数据与模板都就绪时，我们才能顺利拼接出HTML子页面插入到正确的位置上。面对这些复杂的流程，人们不得不发明一些新模式来应对它们。最早被发明出来的是“回调地狱”（callback hell），这应该是一个技能。事实上，几乎JavaScript中的所有异步函数都用到了回调，连续执行几个异步函数的结果就是层层嵌套的回调函数以及随之而来的复杂代码。因此有人说，回调就是程序员的goto语句。

此外，并不是每一个工序都是一帆风顺的，如

果有一个出错了呢，对于JavaScript这样单线程的语言，往往是致命的，必须try...catch，但try...catch语句只能捕捉当前抛出的异常，对后来执行的代码无效。

```
function throwError() {
    throw new Error('ERROR');
}
try {
    setTimeout(throwError, 3000);
} catch (e) {
    alert(e);//这里的异常无法捕获
}
```

　　这些就是本章所要处理的课题。不难理解，domReady、动画、Ajax在骨子里都是同一样东西，假若能将它们抽象成一个东西，显然是非常有用的。

12.1　setTimeout与setInterval

首先我们得深入学习一下这两个API。一般的书籍只是简单介绍它们的用法，没有对它们内在的一些隐秘知识进行描述。它们对我们创建更有用的异步模型非常有用。

（1）如果回调的执行时间大于间隔时间，那么浏览器会继续执行它们，导致真正的间隔时间比原来的大一点。

（2）它们存在一个最小的时钟间隔，在IE6～IE8中为15.6ms，后来精准到10ms，IE10为4ms，其他浏览器相仿。我们可以通过以下函数大致求得此值。

```
function test(count, ms) {
    var c = 1;
    var time = [new Date() * 1];
    var id = setTimeout(function() {
        time.push(new Date() * 1);
        c += 1;
```

```
        if (c <= count) {
            setTimeout(arguments.callee, ms);
        } else {
            clearTimeout(id);
            var tl = time.length;
            var av = 0;
            for (var i = 1; i < tl; i++) {
                var n = time[i] - time[i - 1];
                //收集每次与上一次相差的时间数
                av += n;
            }
            alert(av / count); // 求取平均值
        }
    }, ms);
}
winod.onload = function() {
    var id = setTimeout(function() {
        test(100, 1);
    clearTimeout(id);
    }, 3000);
}
```

具体如表12-1所示。

表**12-1**

Firefox 3.6.3	**Firefox 18.1**	**Chrome 10.53**	**Chrome 23**	**Opera 12.41**	**Safari 5.01**	**IE 8**	**IE 10**
15.59	3.98	3.92	3.6	4.01	4.12	15.91	3.91

但上面的数据很难与官方给出的数值一致，因

为它太容易受外部因素影响，比如电池快没电了，同时打开的应用程序太多了，导致CPU忙碌，这些都会让它的数值偏高。

如果嫌旧版本IE的最短时钟间隔太大，我们或许有办法改造一下setTimeout，利用image死链时立即执行onerror回调的情况进行改造。

```
var orig_setTimeout = window.setTimeout;
window.setTimeout = function (fun, wait) {
    if (wait < 15) {
        orig_setTimeout(fun, wait);
    } else {
        var img = new Image();
        img.onload = img.onerror = function () {
            fun();
        };
        img.src = "data:,foo";
    }
};
```

（3）有关零秒延迟，此回调将会放到一个能立即执行的时段进行触发。JavaScript代码大体上是自顶向下执行，但中间穿插着有关DOM渲染、事件回应等异步代码，它们将组成一个队列，零秒延

迟将会实现插队操作。

（4）不写第二参数，在IE、Firefox中，浏览器自动配时间，第一次配可能给个很大数字，100ms上下，往后会缩小到最小时钟间隔，Safari、Chrome、Opera则多为10ms上下。在Firefox中，setInterval不写第二参数，会当作setTimeout处理，只执行一次。

```
window.onload = function() {
    var a = new Date - 0;
    setTimeout(function() {
        alert(new Date - a);
    });
    var flag = 0;
    var b = new Date,
        text = ""
    var id = setInterval(function() {
        flag++;
        if (flag > 4) {
            clearInterval(id)
            console.log(text)
        }
        text += (new Date - b + " ");
        b = new Date
    })
}
```

（5）标准浏览器与IE10都支持额外参数，从

第3个参数起，作为回调的传参传入！

```
setTimeout(function() {
    alert([].slice.call(arguments));
}, 10, 1, 2, 4);
IE6～IE9可以用以下代码模拟。
if (window.VBArray && !(document.documentMode > 9)) {
    (function(overrideFun) {
        window.setTimeout = overrideFun(window.setTimeout);
        window.setInterval = overrideFun(window.setInterval);
    })(function(originalFun) {
        return function(code, delay) {
            var args = [].slice.call(arguments, 2);
            return originalFun(function() {
                if (typeof code == 'string') {
                    eval(code);
                } else {
                    code.apply(this, args);
                }
            }, delay);
        }
    }
    );
}
```

（6）setTimeout 方法的时间参数若为极端值（如负数、0、或者极大的正数），则各浏览器的处理会出现较大差异，某些浏览器会立即执行。

12.2　Promise诞生前的世界

在Promise被引进到JavaScript世界时，人们通常用以下3个手段解决异步问题。

12.2.1　回调函数callbacks

广义上回调函数的定义为：一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

在JavaScript中，回调函数具体的定义为：函数A作为参数（函数引用）传递到另一个函数B中，并且这个函数B执行函数A，我们就说函数A叫做回

调函数。如果没有名称（函数表达式），就叫做匿名回调函数。因此callback不一定用于异步，一般同步（阻塞）的场景下也经常用到回调，比如要求执行某些操作后执行回调函数。

一个同步（阻塞）中使用回调的例子，目的是在func1代码执行完成后执行func2。

```
function a(callback) {
    callback();
}

function b() {
    console.log('hello callback');
}

a(b); // 注意 b() 作为参数传递给 a() 的时候是不需要带括号的
```

浏览器最早内置的setTimeout与setInteval就是基于回调的思想实现的，Node.js的异步API，都是通过回调实现。

回调是实现异步最朴素的方式。回调函数的优点是简单、容易理解和部署，缺点是不利于代码的

阅读和维护，各个部分之间高度耦合，流程会很混乱，而且每个任务只能指定一个回调函数。

12.2.2　观察者模式observers

观察者模式又叫做发布订阅模式，它定义了一种一对多的关系，让多个观察者对象同时监听某一个主题对象，这个主题对象的状态发生改变时就会通知所有观察着对象。它是由两类对象组成，主题和观察者。主题负责发布事件，同时观察者通过订阅这些事件来观察该主题，发布者和订阅者是完全解耦的，彼此不知道对方的存在，两者仅仅共享一个自定义事件的名称。

在Node.js中通过EventEmitter实现了原生对于这一模式的支持。浏览器中window、document与元素节点自带的事件机制就是基于观察者模式实现的。事实上，我们遍历一下元素节点的原型链，就发现它们的一个原型对象叫做EventTarget!

```
  function PubSub() {
          this.handlers = {};
        }
        PubSub.prototype = {
            // 订阅事件
            on: function(eventType, handler){
                var self = this;
                if(!(eventType in self.handlers)) {
                   self.handlers[eventType] = [];
                }
                self.handlers[eventType].push(handler);
                return this;
            },
             // 触发事件(发布事件)
            emit: function(eventType){
               var self = this;
               var handlerArgs = Array.prototype.slice.call(arg
uments,1);
               for(var i = 0; i < self.handlers[eventType].leng
th; i++) {
                 self.handlers[eventType][i].apply(self,handler
Args);
               }
               return self;
            },
            // 删除订阅事件
            off: function(eventType, handler){
                var currentEvent = this.handlers[eventType];
                var len = 0;
                if (currentEvent) {
                    len = currentEvent.length;
                    for (var i = len - 1; i >= 0; i--){
                        if (currentEvent[i] === handler){
                            currentEvent.splice(i, 1);
                        }
                    }
                }
                return this;
            }
        };

        var pubsub = new PubSub();
        var callback = function(data){
            console.log(data);
        };
```

```
    //订阅事件A
    pubsub.on('A', function(data){
        console.log(1 + data);
    });
    pubsub.on('A', function(data){
        console.log(2 + data);
    });
    pubsub.on('A', callback);

    //触发事件A
    pubsub.emit('A', '我是参数');

    //删除事件A的订阅源callback
    pubsub.off('A', callback);

    pubsub.emit('A', '我是第二次调用的参数');
```

运行结果（见图12-2）。



图12-2

如果将上例改一下。

```
function a(){
   pubsub.emit('b')
```

```
}
pubsub.on('b', function b() {
    console.log('hello callback');
})
```

观察者模式是Gof总结出来的经典设计模式，适应用性非常好，能对调用方与被调用方进行很好的解耦。我们可以用一个pubsub实例当作的软件的“消息中心”，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

12.2.3　事件机制listeners

这个类似于观察者模式。

```
function a(){
   a.emit('b')
}
var pubsub = new PubSub();
for(var i in a){
    a[i] = pubsub[i]
}
a.handlers = {}
//数据自己保存，不共存
a.on('b', function b() {
    console.log('hello callback');
})
```

它的好处是，当目标对象被删除时，不会给整个系统残留一些无用的负担。

观察者模式与事件机制都是适用监听多次同类型的异步行为，不适用于一次性的异步行为。

12.3　JSDeferred里程碑

Deferred是当今最著名的异步模型之一。它原来是Python的Twisted框架中的一个类，后来被Mochikit框架引进来，再后来又被dojo抄去。但那时还一直默默无名，直到JSDeferred诞生。JSDeferred在日本火了几年，人们着手构建CommonJS时，直接以它为蓝图，推出Promise规范。Promise规范又存在A、B、C、D、E多个样本，这都拜那些实现了Promise库的作者所赐，他们都是基于自己的库推出来的。先有实现后有规范，与W3C那帮人的作风相反，反正谁也不服谁。最近，这些Promise库越来越多，最后选取大家认同的几条重新制定出来。这个时期大概拖上几年，因此jQuery就中招了。它最先实现的Deferred/Promise（jQuery 1.5，这是两个东西，其规范里只有一个东西）连Promise A也不算，后来不

得不与Promise A靠近。但是W3C最近敲定的方案与jQuery 1.9的又差得太远。于是jQuery 3为了兼容浏览器内置Promise，只能改大版本号，无法向下兼容。这就是Promise扼要的“血泪史”了，如图12-3所示。

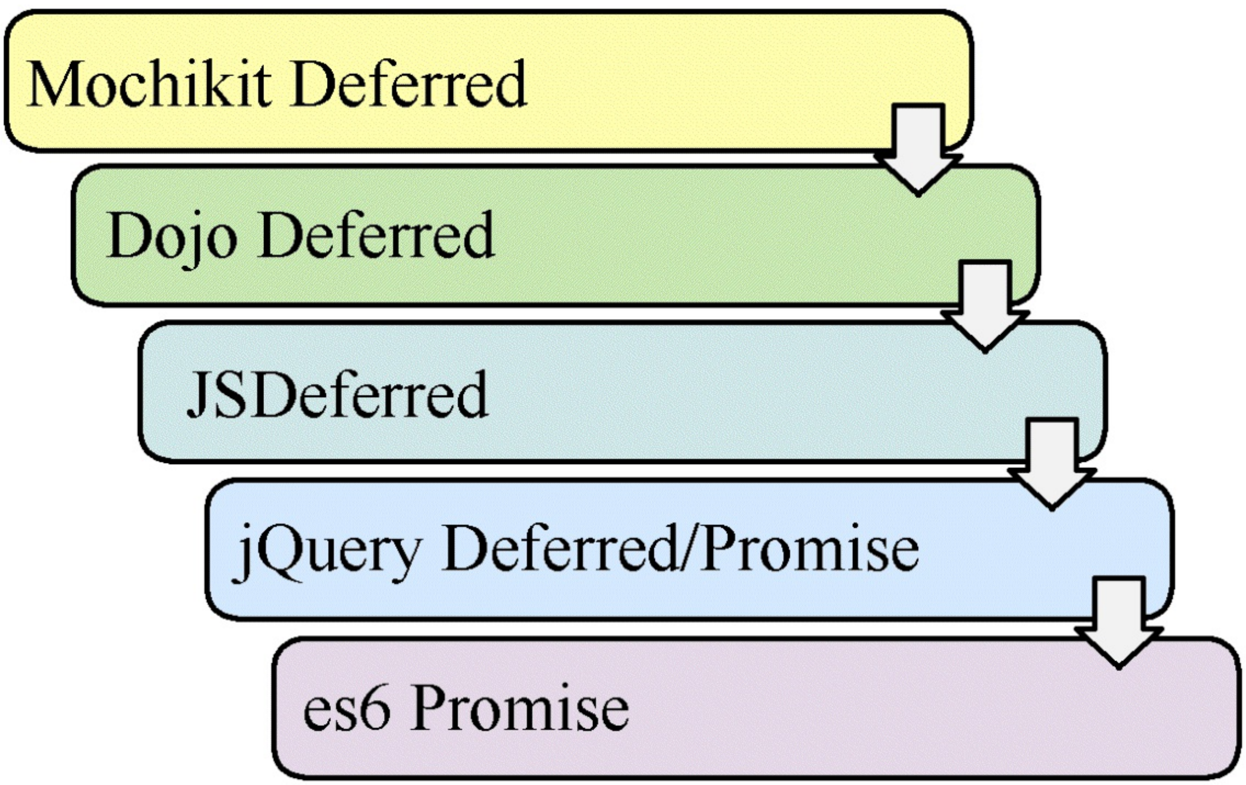


图12-3

Promise发展历史中最重要的一块基石就是JSDeferred，可以说Promise/A+规范的制定则很大

程度地参考了由日本geek cho45发起的jsDeferred项目，追本溯源地了解jsDeferred是十分有必要的。

jsDeferred的特点。

① 内部通过单向链表结果存储成功事件处理函数、失败事件处理函数和链表中下一个Deferred类型对象。

② Deferred实例内部没有状态标识（也就是说Deferred实例没有自定义的生命周期）。

③ 由于Deferred实例没有状态标识，因此不支持成功/失败事件处理函数的晚绑定。

④ Deferred实例的成功/失败事件是基于事件本身的触发而被调用的。

⑤ 由于Deferred实例没有状态标识，因此成功/失败事件可被多次触发，也不存在不变值作为事件

处理函数入参的说法。

Promise/A的特点。

① 内部通过单向链表结果存储成功事件处理函数、失败事件处理函数和链表中下一个Promise类型对象。

② Promise实例内部有状态标识：pending（初始状态）、fulfilled（成功状态）和rejected（失败状态），且状态为单方向移动“pending->fulfilled”“pending->rejected”（也就是Promse实例存在自定义的生命周期，而生命周期的每个阶段具备不同的事件和操作）。

③ 由于Promise实例含状态标识，因此支持事件处理函数的晚绑定。

④ Promise实例的成功/失败事件函数是基于Promise的状态而被调用的。

核心区别

Promises调用成功/失败事件处理函数的两种流程。

① 调用resolve/reject方法尝试改变Promise实例的状态，若成功改变其状态，则调用Promise当前状态相应的事件处理函数（类似于触发onchange事件）。

② 通过then方法进行事件绑定，若Promise实例的状态不是pending，则调用Promise当前状态相应的事件处理函数。

由上述可以知道Promises的成功/失败事件处理函数均基于Promise实例的状态而被调用，而非成功/失败事件。

jsDeferred调用成功/失败事件处理函数的流程：

调用call/fail方法触发成功/失败事件，则调用相应的事件处理函数。

因此jsDeferred的是基于事件的。

下列内容均为大概介绍API接口，具体用法请参考官网。

1. 构造函数

Deferred，可通过 new Deferred()或Deferred()两种方式创建Deferred实例。

```
var defer = Deferred();//或new Deferred()
    //创建一个Deferred对象
  defer.next(function () {
      console.log('ok');
  }).error(function (text) {
      console.log(text);//=> test
  }).fail('test');
```

2. 实例方法

Deferred.prototype.next(fn)，绑定成功事

件处理函数，返回一个新的Deferred实例。在没有调用`Deferred.prototype.call`前这个事件处理函数并不会执行。

```
var deferred = Deferred();
deferred.next(function (value) {
    console.log(value); // => aaa
}).call('aaa');
```

`Deferred.prototype.error(fn)`，绑定失败事件处理函数，返回一个新的Deferred实例。在没有调用`Deferred.prototype.fail`前这个事件处理函数并不会执行。

```
var deferred = Deferred();
deferred.error(function () {
    console.log('error');// => error
}).fail();
```

`Deferred.prototype.call(val*)`，触发成功事件，返回一个新的Deferred实例。

Deferred.prototype.fail(val*)，触发失败事件，返回一个新的Deferred实例。

3. 静态属性

Deferred.ok，默认的成功事件处理函数。

Deferred.ng，默认的失败事件处理函数。

Deferred.methods，默认的向外暴露的静态方法（供Deferred.define方法使用）。

```
Deferred.methods = ["parallel", "wait", "next", "call", "loop",
 "repeat", "chain"];
```

4. 静态方法

Deferred.define(obj, list)，暴露静态方法到obj上，无参的情况下obj是全局对象，侵入性极强，但使用方便。list是一组方法，这组方法会同时注册到obj上。

Deferred.call({Function} fn [, arg]*)，创建一个Deferred实例并且触发其成功事件。

```
//无参，侵入式，默认全局对象，浏览器环境为window
Deferred.define();
//静态方法入next被注册到了window下
next(function () {
    console.log('ok');
});
var defer = {};
//非侵入式，Deferred的静态方法注册到了defer对象下
Deferred.define(defer);
defer.next(function () {
    console.log('ok');
});
```

Deferred.next({Function} fn)，创建一个Deferred实例并且触发其成功事件，其实就是无法传入参到成功事件处理函数的 Deferred.call()。

```
Deferred.define();
next(function () {
    console.log('ok');
});
console.log('hello,world!');// => 先输出
//上面的代码等同于下面的代码
call(function () {
   console.log('ok');
});
console.log('hello,world!');// => 先输出
```

`Deferred.wait(sec)`，创建一个Deferred实例，并等待sec（秒）后触发其成功事件，下面的代码首先弹出“Hello,”，2秒后弹出“World!”。

```
next(function () {
    alert('Hello,');
    return wait(2);//延迟2s后执行
}).
next(function (r) {
    alert('World!');
});
console.log('hello,world!');// => 先输出
```

`Deferred.loop(n, fun)`，循环执行n次fun，并将最后一次执行fun()的返回值作为Deferred实例成功事件处理函数的参数，同样loop中循环执行的fun()也是异步的。

```
loop(3, function () {
    console.log(count);
    return count++;
}).next(function (value) {
    console.info(value);// => 2
});
//上面的代码也是异步的（无阻塞的）
console.info('aaa');
```

Deferred.parallel(dl[,fn]*)，把参数中非Deferred对象均转换为Deferred对象（通过Deferred.next()），然后并行触发dl中的Deferred实例的成功事件。

当所有Deferred对象均调用了成功事件处理函数后，返回的Deferred实例则触发成功事件，并且所有返回值将被封装为数组作为Deferred实例的成功事件处理函数的入参。

parallel()强悍之处在于它的并归处理，它可以将参数中多次的异步最终并归到一起，这一点在JavaScript ajax嵌套中尤为重要，例如同时发送两条ajax请求，最终parallel()会并归这两条ajax返回的结果。

parallel()进行了3次重载。

- parallel(fn[,fn]*)，传入Function类型的参数，允

许多个。

- parallel(Array)，给定一个由Function组成的Array类型的参数。
- parallel(Object)，给定一个对象，由对象中所有可枚举的Function构建Deferred。

图12-4演示了Deferred.parallel的工作模型，它可以理解为合并了3次ajax请求。

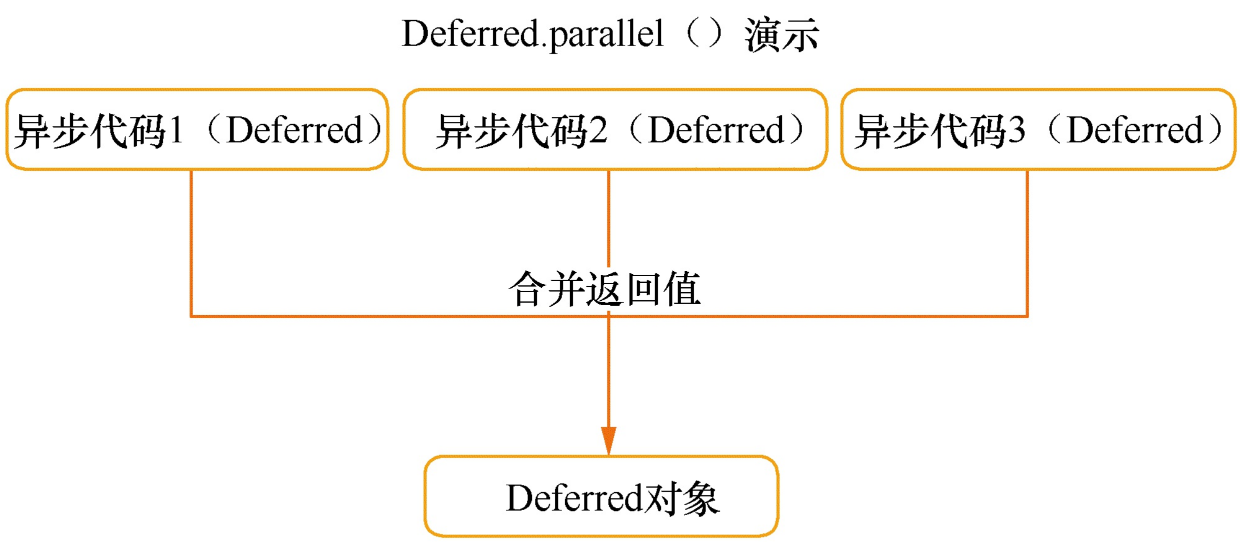


图12-4

```
Deferred.define();
parallel(function () {
    //等待2秒后执行
    return wait(2).next(function () { return 'hello,'; });
}, function () {
```

```
    return wait(1).next(function () { return 'world!' });
}).next(function (values) {
    console.log(values);// =>  ["hello,", "world!"]
});
```

当parallel传递的参数是一个对象的时候，返回值则是一个对象。

```
parallel({
     foo: wait(1).next(function () {
         return 1;
     }),
     bar: wait(2).next(function () {
         return 2;
     })
}).next(function (values) {
     console.log(values);// =>  Object { foo=1, bar=2 }
});
```

和jQuery.when()如出一辙。

Deferred.earlier(dl)，当参数中某一个Deferred对象调用了成功处理函数，则终止参数中其他Deferred对象的触发的成功事件，返回的Deferred实例则触发成功事件，并且那个触发成功事件的函数返回值将作为Deferred实例的成功事件

处理函数的入参。

注意：Deferred.earlier()并不会通过Deferred.define(obj)暴露给obj，它只能通过Deferred.earlier()调用。

Deferred.earlier()内部的实现和Deferred.parallel()大同小异，但值得注意的是参数，它接受的是Deferred，而不是parallel()的Function。

- Deferred.earlier(Deferred[,Deferred]*)，传入Deferred类型的参数，允许多个。
- Deferred.earlier(Array)，给定一个由Deferred组成的Array类型的参数。
- Deferred.earlier(Object)，给定一个对象，由对象中所有可枚举的Deferred构建Deferred。

```
Deferred.define();
Deferred.earlier(
    wait(2).next(function () { return 'cnblog'; }),
```

```
    wait(1).next(function () { return 'aaa' })//1s后执行成功
).next(function (values) {
    console.log(values);// 1s后 => [undefined, "aaa"]
});
```

Deferred.repeat(n, fun)，循环执行fun方法n次，若fun的执行事件超过20毫秒则先将UI线程的控制权交出，等一会儿再执行下一轮的循环。

Deferred.chain(args)，chain()方法的参数比较独特，可以接受多个参数，参数类型可以是：Function、Object、Array。chain()方法比较难懂，它是将所有的参数构造出一条Deferred方法链。例如Function类型的参数。

```
Deferred.define();
chain(
    function () {
        console.log('start');
    },
    function () {
        console.log('linkFly');
    }
);
//等同于
next(function () {
    console.log('start');
}).next(function () {
```

```
    console.log('linkFly');
});
```

它通过函数名来判断函数。

```
chain(
    //函数名!=error，则默认为next
    function () {
        throw Error('error');
    },
    //函数名为error
    function error(e) {
        console.log(e.message);
    }
);

//等同于
next(function () {
    throw Error('error');
}).error(function (e) {
    console.log(e.message);
});
```

也支持Deferred.parallel()的方式。

```
chain(
      [
          function () {
              return wait(1);
          },
          function () {
              return wait(2);
          }
      ]
  ).next(function () {
```

```
      console.log('ok');
  });

//等同于
 Deferred.parallel([
     function () {
         return wait(1);
     },
     function () {
         return wait(2);
     }
 ]).next(function () {
     console.log('ok');
 });
```

`Deferred.connect(funo, options)`，将一个函数封装为Deferred对象，其目的是融入现有的异步编程。

Deferred.connect()有两种重载。

- Deferred.connect(target,string)，把target上名为string指定名称的方法包装为Deferred对象。
- Deferred.connect(function,Object)，Object至少要有一个属性：target。以target为this调用function方法，返回的是包装后的方法，该方法返回

Deferred对象。

给包装后的方法传递的参数，会传递给所指定的function。

```
var timeout = Deferred.connect(setTimeout, { target: window, ok
: 0 });
timeout(1).next(function () {
     alert('after 1 sec');
});
//另外一种传参
var timeout = Deferred.connect(window, "setTimeout");
timeout(1).next(function () {
    alert('after 1 sec');
});
```

Deferred.retry(retryCount, funcDeferred, options)，调用retryCount次funcDeffered方法（返回值类型为Deferred），直到触发成功事件或超过尝试次数为止。options参数是一个对象，{wait:number}指定每次调用等待的秒数。

注意：Deferred.retry()并不会通过Deferred.define(obj)暴露给obj，它只能通过

Deferred.retry()调用。

```
Deferred.define();
Deferred.retry(3, function (number) {//Deferred.retry()方法是--i的方式实现的
    console.log(number);
    return Deferred.next(function () {
        if (number ^ 1)//当number!=1的时候抛出异常，表示失败，number==1的时候则让它成功
            throw new Error('error');
    });
}).next(function () {
    console.log('linkFly');//=>linkFly
});
```

Deferred.register(name, fn)，将静态方法附加Deferred. prototype上。

核心源码解读：

构造函数部分。

```
function Deferred () { return (this instanceof Deferred) ? this.init() : new Deferred() }
// 默认的成功事件处理函数
Deferred.ok = function (x) { return x };
// 默认的失败事件处理函数
Deferred.ng = function (x) { throw  x };
Deferred.prototype = {
    // 初始化函数
    init : function () {
    this._next    = null;
```

```
    this.callback = {
        ok: Deferred.ok,
        ng: Deferred.ng
    };
    return this;
}};
```

触发回调部分。

```
Deferred.prototype.call = function (val) { return this._fire("o
k", val) };
Deferred.prototype.fail = function (err) { return this._fire("n
g", err) };
Deferred.prototype._fire = function(okng, value){
    var next = "ok";
    try {
        // 调用当前Deferred实例的事件处理函数
        value = this.callback[okng].call(this, value);
    } catch (e) {
        next = "ng";
        value = e;
        if (Deferred.onerror) Deferred.onerror(e);
    }
    if (Deferred.isDeferred(value)) {
        // 若事件处理函数返回一个新Deferred实例，则将新Deferred实例的链
表指针指向当前Deferred
        // 实例的链表指针指向
        // 这样新Deferred实例的事件处理函数就会先与原链表中其他Deferred
实例的事件处理函数被调用。
        value._next = this._next;
    } else {
        if (this._next) this._next._fire(next, value);
    }
    return this;
};
```

添加回调部分。

```
Deferred.prototype.next = function (fun) { return this._post("o
k", fun) };
Deferred.prototype._post = function (okng, fun) {
    // 创建一个新的Deferred实例，插入Deferred链表尾，并将事件处理函数绑
定到新的Deferred上
    this._next = new Deferred();
    this._next.callback[okng] = fun;
    return this._next;
};
Deferred.next =
    Deferred.next_faster_way_readystatechange ||
    Deferred.next_faster_way_Image ||
    Deferred.next_tick ||
    Deferred.next_default;
```

Deferred.next 是jsDeferred最出彩的地方了，也是后续其他方法的实现基础，它的功能是创建一个新的Deferred对象，并且异步执行该Deferred对象的call方法来触发成功事件。针对运行环境的不同，它提供了相应的异步调用的实现方式并作出降级处理。

由浅入深，我们先看看使用setTimeout实现异步的Deferred.next_default 方法（存在最小时间

精度的问题）。

```
Deferred.next_default = function (fun) {
    var d = new Deferred();
    var id = setTimeout(function () { d.call() }, 0);
    d.canceller = function () { clearTimeout(id) };
    if (fun) d.callback.ok = fun;
    return d;
};
```

然后是针对nodejs的`Deferred.next_tick`方法。

```
Deferred.next_tick = function (fun) {
    var d = new Deferred();
    // 使用process.nextTick来实现异步调用
    process.nextTick(function() { d.call() });
    if (fun) d.callback.ok = fun;
    return d;
};
```

然后就是针对现代浏览器的`Deferred.next_faster_way_Image`方法。

```
Deferred.next_faster_way_Image = function (fun) {
    var d = new Deferred();
    var img = new Image();
    var handler = function () {
        d.canceller();
```

```
        d.call();
    };
    img.addEventListener("load", handler, false);
    img.addEventListener("error", handler, false);
    d.canceller = function () {
        img.removeEventListener("load", handler, false);
        img.removeEventListener("error", handler, false);
    };
    // 请求一个无效data uri scheme导致马上触发load或error事件
    // 注意：先绑定事件处理函数，再设置图片的src是个良好的习惯。因为设置im
g.src属性后就会马上发起
    // 请求，假如读的是缓存那有可能还未绑定事件处理函数，事件已经被触发了
    img.src = "data:image/png," + Math.random();
    if (fun) d.callback.ok = fun;
    return d;
};
```

根据JSDeferred官方的数据，用上这个后至少比原有的setTimeout异步方式快上700%以上，如图12-5所示。

环境	setTimeout	faster_way	%
Opera 9.5 (Mac)	80sec	1.7sec	4706%
Internet Explorer (Win), calc time	120sec	6sec	2000%
Safari 3.1 (Mac)	80sec	6sec	1300%
Google Chrome (Win)	31sec	2.4sec	1200%
Firefox 3.1b2 (Mac)	83sec	10sec	830%
Internet Explorer (Win)	120sec	17sec	700%

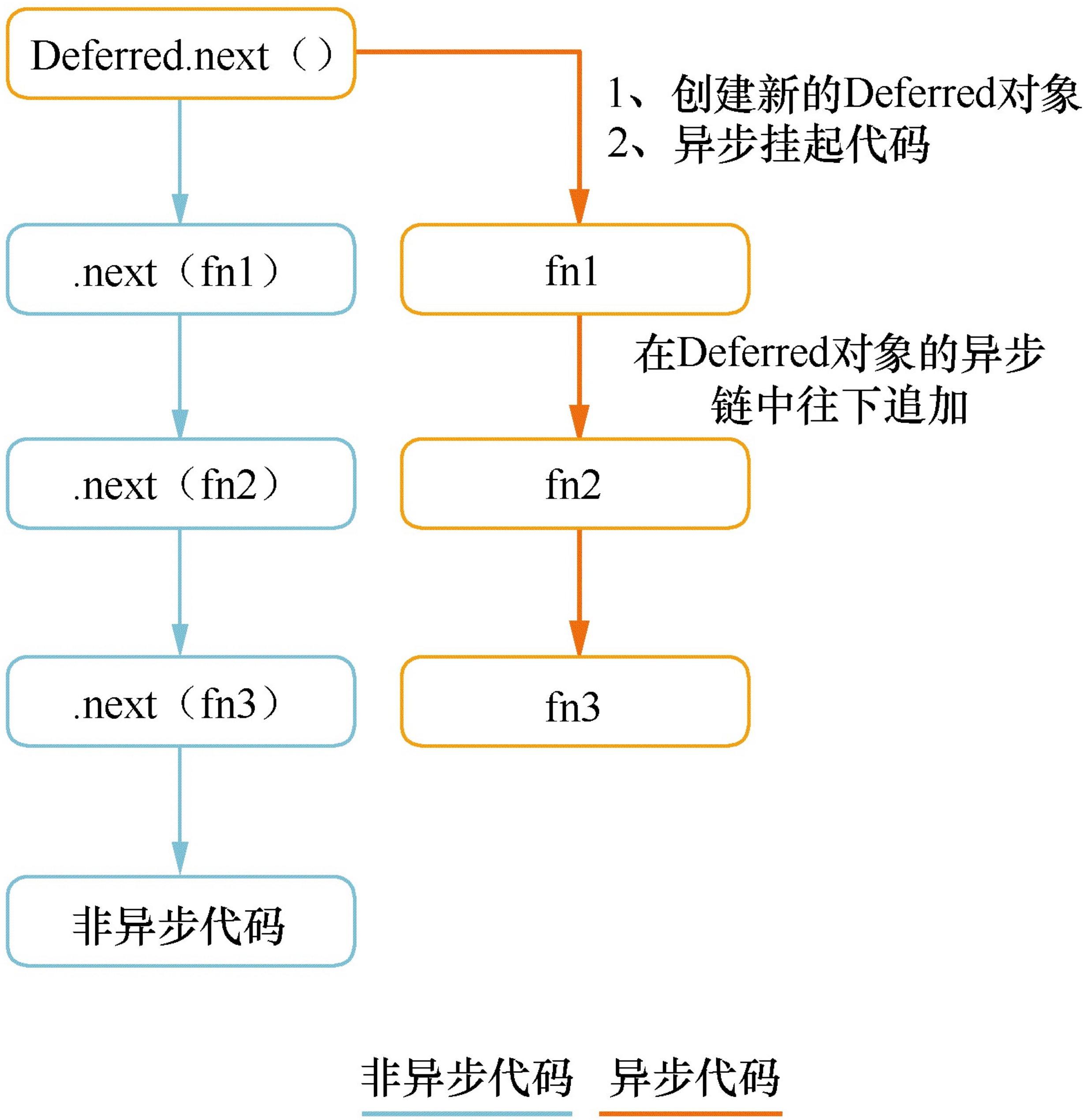
图12-5

最后就是针对IE5.5～IE8的Deferred.next*faster*way_readystatechange方法。

```
Deferred.next_faster_way_readystatechange = ((typeof window ===
 'object') && (location. protocol == "http:") && !window.opera
&& /\bMSIE\b/.test(navigator.userAgent)) && function (fun) {
    var d = new Deferred();
    var t = new Date().getTime();
    /* 原理：
            由于浏览器对并发请求数作出限制（IE5.5~IE8为2~3,IE9+和现代
浏览器为6），
            因此当并发请求数大于上限时，会让请求的发起操作排队执行，导致延
时更严重了。
       实现手段：
            以150毫秒为一个周期，每个周期以通过setTimeout发起的异步执行
作为起始，
            周期内的其他异步执行操作均通过script请求实现。
            （若该方法将在短时间内被频繁调用，可以将周期频率再设高一些，如
100毫秒）
    */
    if (t - arguments.callee._prev_timeout_called < 150) {
        var cancel = false;
        var script = document.createElement("script");
        script.type = "text/javascript";
        // 采用无效的data uri sheme马上触发readystate变化
        script.src  = "data:text/javascript,";
        script.onreadystatechange = function () {
            // 由于在一次请求过程中script的readystate会变化多次，因此
通过cancel标识来保证仅调
            // 用一次call方法
            if (!cancel) {
                d.canceller();
                d.call();
            }
        };
        d.canceller = function () {
            if (!cancel) {
                cancel = true;
                script.onreadystatechange = null;
                document.body.removeChild(script);
            }
        };
        // 不同于img元素，script元素需要添加到dom树中才会发起请求
        document.body.appendChild(script);
    } else {
        arguments.callee._prev_timeout_called = t;
        var id = setTimeout(function () { d.call() }, 0);
        d.canceller = function () { clearTimeout(id) };
```

```
    }
    if (fun) d.callback.ok = fun;
    return d;
};
```

图12-6粗略演示了jsDeferred的工作流程。



Deferred.next（）
1、创建新的Deferred对象
2、异步挂起代码
.next（fn1）
fn1
在Deferred对象的异步
链中往下追加
.next（fn2）
fn2
.next（fn3）
fn3
非异步代码
非异步代码
异步代码

图12-6

12.4 jQuery Deferred宣教者

jQuery的异步模型是从jQuery 1.5开始搞的，那时Promise还没有定稿，因此jQuery的Promise与现在的很不一样。它最底层是一个叫Callbacks的对象，实际上就是一个特殊的列表，可以轻松实现观察者的功能。然后3个Callbacks合成1个Deferred，外加1个when函数，就是实现JSDeferred那种并归多个异步操作的效果。当时马上就投入Ajax模块的改造，立即让此模块难读十倍以上。

但推出以来，它一直被雪藏着，没有在文档中露脸，直到jQuery1.52正式独立成一个模块。Deferred模块是当时最难读的代码，因为Promise本来就是一个很学术性的东西，接口没有固化一下，Deferred那些reject、resolve、then、pipe方法名，让人摸不着头脑，如果把reject 更名为fireError, resolve更名为fireSuccess，其受众面更些。

如果深究Deferred的实现，它与JSDeferred差不多，将一个个异步操作封装成一个个对象，然后设法连在一起。为了保证外界不改变其状态，jQuery更是使用闭包，将状态封闭起来，只有通过几个特别方法（resolve、reject）才能修改它。一旦修改，这个状态就不可更改。这也是就是Promise规范的要点之一。

由于先于Promise规范形成，它与真正Promise是有差异的。规范中，只有一个Promise对象。而jQuery是通过Deferred与Promise两个对象实现的。经常跑到你眼前的是Deferred，它能同时添加各种回调与触发各种回调。此外，还有一个叫promise的对象，它是Deferred的某些API产生的对象，看起来能实现jQuery的链式操作。但jQuery的链式操作总是返回jQuery对象，而那些方法则是生成promise对象，它是一个只读的Deferred，只能添加回调，不能触发回调执行。换言之，它们不能修改异步操作

的状态。并且，同一个Deferred对象的链式操作API总是返回同一个promise。John Resig认为这样更节能，封装性更好。其实这违背了规范，Promise规范要求链式操作API总是产生一个新的Promise。由于时间的错位，我们不能责怪John Resig，但意味着以后为了遵循Promise标准API，它的API需要调整了。

最惨的API是then 方法，一共调整了3次。

```
// version added: 1.5, removed: 1.8
deferred.then( doneFn, failFn )
// version added: 1.7, removed: 1.8
deferred.then( doneFn, failFn [, progressFn ] )
// version added: 1.8
deferred.then( doneFn [, failFn ] [, progressFn] )
```

后来连他们也不好意思了，于是推出Deferred.pipe方法。到了jQuery3.0, Deferred.then又再次反客为主，成为文档上推荐使用的API，pipe方法只是留作兼容用。

为了方便学习jQuery，明白这些API是什么意思，我们把它与早期的Deferred库放在一起，就一目了然，如表12-2所示。

表**12-2**

	jQuery.ajax	**JSDeferred**	**jQuery Deferred**	**Promise**
添加一个成功回调	addCallback	next	promise.done	then
添加一个失败回调	addErrback	error	promise.fail	then
添加一组回调	addBoth		promise.then	then
触发成功列队	callback	call	resolve/resolveWith	resolve
触发失败列队	errback	fail	reject/rejectWith	reject
取消	cancel	cancal		
询问当前状态	state			
并归结束	DeferedList	parallel	when	Promise.all

此外jQuery的还有几个特别的API，如表12-3所

示。

- 不改变当前状态的添加回调与触发回调Promise/A+规范正在密谋将progress与notify它们标谁化。
- 总是触发的回调，它们通过always方法添加，Promise/A+称之为**finally** 。

表**12-3**

	jQuery.ajax	**jQuery Deferred**
添加一个成功回调	success	done
添加一个失败回调	error	fail
添加一个总是执行的回调	complete	always

此外，需要提一下then方法，它不等同于Promise的then，并且它的实现一直在变，因此jQuery搞了一个pipe方法来取替then。直到jQuery

3.0，Deferred.then才完全等价Promise.then。

之前说过，jQuery的Deferred是基于3个Callbacks。我们粗略看一下其结构。

```
$.Deferred = function( func ) {
var tuples = [
   [ "notify", "progress", jQuery.Callbacks( "memory" ),
            jQuery.Callbacks( "memory" ), 2 ],
   [ "resolve", "done", jQuery.Callbacks( "once memory" ),
            jQuery.Callbacks( "once memory" ), 0, "resolved" ],
   [ "reject", "fail", jQuery.Callbacks( "once memory" ),
            jQuery.Callbacks( "once memory" ), 1, "rejected" ]
    ],
    state = "pending",
    promise = {
        state: function() {
            return state;
        },
        always: function() {
            deferred.done( arguments ).fail( arguments );
            return this;
        },
        "catch": function( fn ) {
            return promise.then( null, fn );
        },
        then: function(){
        },
        when: function(){
        },
        promsie: function(){
        }
    }
    var deferred = {}
    promise.promise( deferred );
    return deferred

}
```

简单转换一下，其他相当于。

```
var resolveCb = jQuery.Callbacks("once memory");
var rejectCb = jQuery.Callbacks("once memory");
var notifyCb = jQuery.Callbacks("memory");

done        = resolveCb.add
resolveWith = resolveCb.fireWith
resolve     = resolveCb.fireWith.bind(this)

fail        = rejectCb.add
rejectWith  = rejectCb.fireWith
reject     = rejectCb.fireWith.bind(this)

progress    = notifyCb.add
notifyWith  = notifyCb.fireWith
notify  = notifyCb.fireWith.bind(this)
```

Callbacks合并Deferred后，其一些API还可以继续使用。

```
var dd = $.Deferred();
dd.done(function (name) {
    console.log(name, 1);
}).done(function (name) {
    console.log(name, 2);
});
dd.resolve('标准API');
//其实就相当于
resolveCb.add(function (name) {
    console.log(name, 1);
}).add(function (name) {
    console.log(name, 2);
```

```
});
resolveCb.fire('残留API');
```

当Deferred执行完resolve以后，同时会调用rejectCb.disable和notifyCb.lock。当Deferred执行完reject以后，同时会调用resolveCb.disable和notifyCb.lock。notifyCb就是一个单纯的$.Callbacks，但是它的状态会受到resolve/reject的影响。

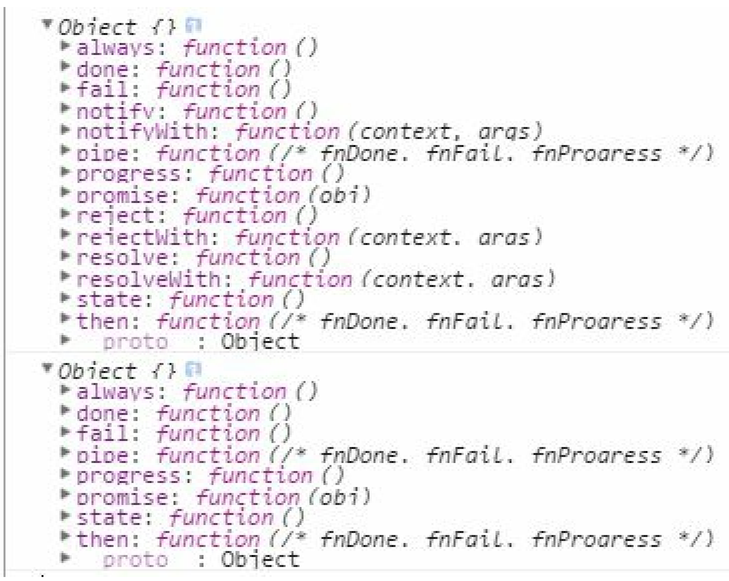
resolveCb和rejectCb两者之间是相互限制的，一旦两者中的某一个fire了，另一个就会被disable。通过这种方式来达到唯一状态。

最后看一下 jQuery Deferred与ES6 Promise的区别，其实最开始浏览器厂商也打算抄jQuery的API，最后被CommonJS 那帮人压制住了，变成现在这个模型。

```
//jQuery
```

```
var deferred = $.Deferred();
var promise = deferred.promise();
//Promsie浏览器早期内置版本
var deferred2 = Promise.defer();
var promise2= defered.promise;
//es6标准化的Promise
var promise3 = new Promise(function( resolve, reject){})

console.log(deferred, promise)
```



由于jQuery的异步模型带了“太多私货”，因此不再浪费笔墨研究它了，但它的确为大众接受Promsie做了许多前期宣传工作。当Promsie被浏览器内置时，前端开发人员已经用它得心应手了。

12.5　es6 Promise第一个标准模型

Promise在2014年4月率先被Firefox 29实现，紧接着Chrome 32、微软Edge都支持原生Promise。当然，在此之前，Node.js0.12层已经直接支持，异步的需求在后端更为强劲。Promise是JavaScript第一个标准的异步模型。受到这鼓舞，浏览器一些新API也使用Promise实现。

- Battery API。
- fetch API（XMLHttpRequest的取代者）。
- ServiceWorder API。
- await/async 的内部实现。

Promise是一个包含传递信息与状态的对象，拥有以下两个特点。

（1）对象的状态不受外界影响。Promise对象

代表一个异步操作，有3种状态：Pending（进行中）、Resolved（已完成，又称Fulfilled）和Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，如图12-7所示。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从Pending变为Resolved和从Pending变为Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函

数。

Promise在前端主要是解决Ajax的异步问题，主要有3种情况。

（1）由于Ajax是异步的，所有依赖Ajax返回结果的代码必需写在Ajax回调函数中。这就不可避免地形成了嵌套，Ajax等异步操作越多，嵌套层次就会越深，代码可读性就会越差。

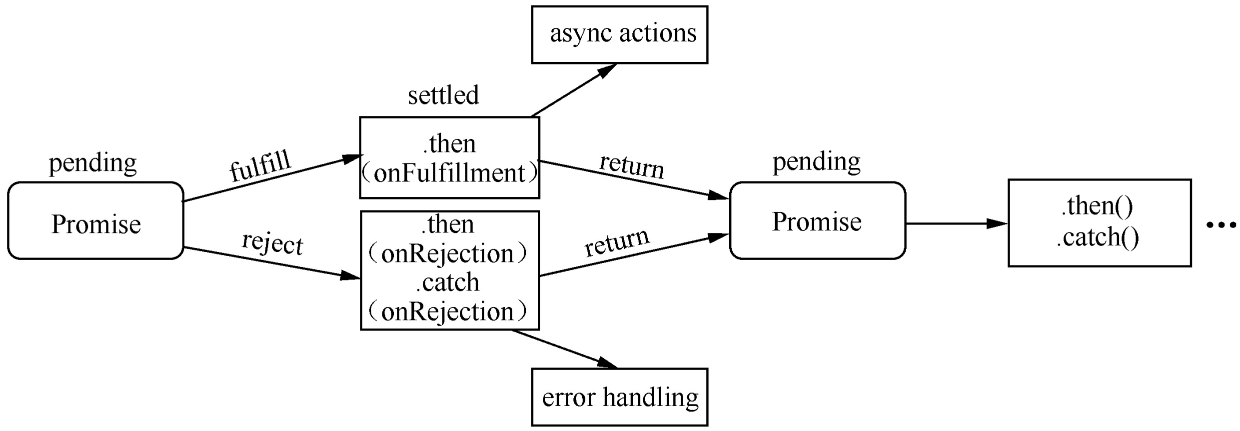


图12-7

```
$.ajax({
    url: url,
    data: dataObject,
    success: function(){
    console.log("I depend on ajax result.");
    },
    error: function(){}
});

console.log("I will print before ajax finished.");
```

（2）我们向后端请求数据时，后端不能通过一个接口帮我们搞定，需要从A接口得到一部分数据，然后再根据A数据的某个属性再从B接口得到剩下的数据。更有甚者，还有取C接口、D接口的，因此会造成Ajax的回调里套着另一个Ajax，一

层层下去就诞生著名的回调地狱 问题，如图12-8所示。

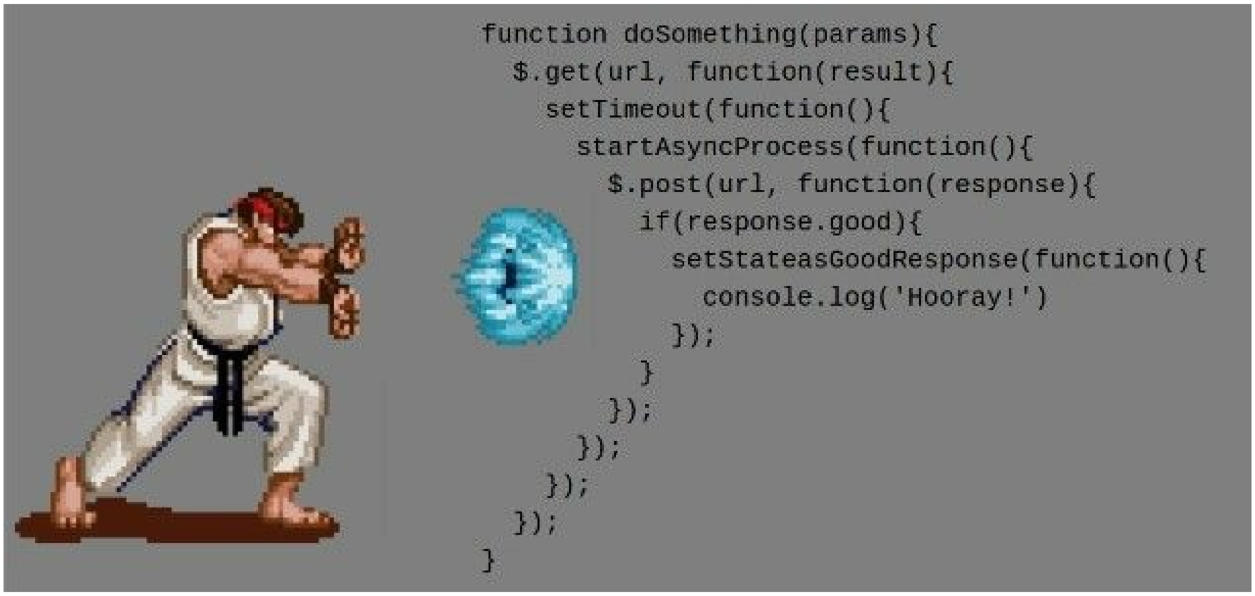


图12-8

```
$.ajax({
    url: url1,
    success: function(data){
        $.ajax({
            url: url2,
            data: data,
            success: function(data){
                $.ajax({
                    //...
                });
            }
        });
    }
});
```

（3）某个操作，必须等于前面两个Ajax回来才能执行。比如说，一个要请求模板，另一个要请求数据。这逼使程序员自己写计时器，回来一个就减1，等于0时执行最后的回调。

```
function loadImg(url, cb) {
    var img = new Image();
    img.src = url;
    img.onload = cb;
}

function loadImages(urlArr, afterAllLoadedFunc) {
    var count = urlArr.length;
    var loadedCount = 0;

    for (var i = count - 1; i >= 0; i--) {
        loadImg(urlArr[i], function () {
            loadedCount += 1;
            if (count === loadedCount) {
                afterAllLoadedFunc();
            }
        });
    }
}

loadImages(['./xx.jpg', './yy.jpg', './zz.jpg'], function () {
    alert('all imgs have been loaded');
});
```

精简一下，可以描述为。

（1）处理异步回调。

（2）多个异步回调的串行处理。

（3）多个异步回调的并行处理。

首先Promise的API是链式，所有同步的代码，可以写到其then方法中。

```
var promise = new Promise(function (resolve, reject) {
    console.log('begin do something');
    if (Math.random() * 10.0 > 5) {
        console.log(" run success");
        resolve();//正常执行
    } else {
        console.log(" run failed");
        reject();//发生异常时
    }
});

promise.then(function () {
    console.log(' resolve from promise');
}, function () {
    console.log(' reject from promise');
});
```

执行结果一：

```
begin do something
 run success
 resolve from promise
```

执行结果二：

```
begin do something
 run failed
 reject from promise
```

让值在多个回调里进行流水化处理。

```
function double(value) {
    return value * 2;
}
function increment(value) {
  return value+9
}
function output(value) {
    console.log(value);// => (1 + 9) * 2
}

var promise = Promise.resolve(1);
promise
    .then(increment)
    .then(double)
    .then(output)
    .catch(function(error){
        console.error(error);
    });
```

处理Ajax回调。

```
function get(url) {
  // Return a new promise.
  return new Promise(function(resolve, reject) {
```

```
    // Do the usual XHR stuff
    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      // This is called even on 404 etc
      // so check the status
      if (req.status == 200) {
        // Resolve the promise with the response text
        resolve(req.response);
      }
      else {
        // Otherwise reject with the status text
        // which will hopefully be a meaningful error
        reject(Error(req.statusText));
      }
    };

    // Handle network errors
    req.onerror = function() {
      reject(Error("Network Error"));
    };

    // Make the request
    req.send();
  });
}

// Use it!
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.error("Failed!", error);
});
```

多个异步回调的串行处理，直接在回调里面返回一个Promise。

```
function Pro1(orderId){
    return new Promise(function(resolve, reject) {
        setTimeout(function(){
            var orderInfo = {
                orderId: orderId,
                productIds: ['123', '456']
            }
            resolve(orderInfo.productIds)
        }, 300)
    })
}
function Pro2(productIds){
    return new Promise(function(resolve, reject) {
        setTimeout(function(){
            var products = productIds.map(function(productId){
                return {
                    productId: productId,
                    name: '衣服'
                }
            })
            resolve(products)
        }, 300)
    })
}
//调用
Pro1('abc123')
.then(function(productIds){
    console.log('商品id',productIds)
    return Pro2(productIds)
})
.then(function(products){
    console.log('商品详情',products)
})
.catch(function(err){
    throw new Error(err)
})
```

多个异步回调的并行处理，使用Promise.all，它需要接受多个Promise作为参数。

```
Promise.all([promise1, promise2]).then(function(results) {
    // Both promises resolved
})
.catch(function(error) {
    // One or more promises was rejected
});
```

如果使用fetch来请求数据，它返回的直接是带数据的Promise。

```
var request1 = fetch('/users.json');
var request2 = fetch('/articles.json');

Promise.all([request1, request2]).then(function(results) {
    // Both promises done!
});
```

此外还有一个`Promise.race`，它相当于JSDeferred的earlier。只要有一个Promise的状态被改变，即被调用了resolve或reject，它就会执行Promise.race。

```
var req1 = new Promise(function(resolve, reject) {
    // A mock async action using setTimeout
    setTimeout(function() { resolve('First!'); }, 8000);
});
var req2 = new Promise(function(resolve, reject) {
    // A mock async action using setTimeout
```

```
    setTimeout(function() { resolve('Second!'); }, 3000);
});
Promise.race([req1, req2]).then(function(one) {
    console.log('Then: ', one);
}).catch(function(one, two) {
    console.log('Catch: ', one);
});

// From the console:
// Then: Second!
```

更多API，可以到MDN上搜，总共就reject、resolve、then、catch、all、race这几个方法，与我们前几节学到的异步库的API大同小异，只是改个名而已。

下面我们来实现一下Promise。Promise是一个普通的jS函数，因此用ES3的语法也能将它实现出来，并运行于IE6这样旧版本的浏览器。

12.5.1　构造函数：Promise (executor)

Promise是一个构造函数，它需要传入一个executor方法。

```
var nativePromise = window.Promise
if (/native code/.test(nativePromise)) {//判定浏览器是否支持原生Pro
mise
    module.exports = nativePromise
} else {
    var RESOLVED = 0
    var REJECTED = 1
    var PENDING = 2

    //实例化Promise
    function Promise(executor) {
        this.state = PENDING
        this.value = undefined
        this.deferred = []
        var promise = this
        try {
            executor(function (x) {
                promise.resolve(x)
            }, function (r) {
                promise.reject(r)
            })
        } catch (e) {
            promise.reject(e)
        }
    }
    //略……
 }
```

12.5.2　Promise.resolve/reject

提供另一种手段，快速实例化一个Promise，并已经传入参数与触发Promise链的执行。

```
    Promise.resolve = function (x) {
        return new Promise(function (resolve, reject) {
            resolve(x)
```

```
        })
    }
    Promise.reject = function (r) {
        return new Promise(function (resolve, reject) {
            reject(r)
        })
}
```

12.5.3　Promise.all/race

它们相当于JSDeferred的parallel/earlier，用于数据处理的并发或竞争。它们都要求传一个Promise数组作为参数，然后返回一个新的Promise。

```
 Promise.all = function (iterable) {
     return new Promise(function (resolve, reject) {
         var count = 0, result = []
         if (iterable.length === 0) {
             resolve(result)
         }
         function resolver(i) {
             return function (x) {
                 result[i] = x //收集所有结果
                 count += 1
                 //数量等于传参个数时才执行新生成的Promise
                 if (count === iterable.length) {
                     resolve(result)
                 }
             }
         }

         for (var i = 0; i < iterable.length; i += 1) {
             Promise.resolve(iterable[i]).then(resolver(i), rej
ect)
         }
```

```
    })
}
```

Promise.race只要有一个promise对象进入FulFilled或者Rejected状态的话，就会继续进行后面的处理。换言之，谁的异步时间最短，谁就会被处理。

```
Promise.race = function (iterable) {
    return new Promise(function (resolve, reject) {
        for (var i = 0; i < iterable.length; i += 1) {
            Promise.resolve(iterable[i]).then(resolve, reject)
        }
    })
}
```

12.5.4 Promise#then/catch

它的这两个原型方法都是用来添加回调，然后返回新的Promise，伪装成Promise。

```
var p = Promise.prototype
//构建Promise列队,并将要执行的函数与传参存储到最初的Promise.deferred数组中
p.then = function then(onResolved, onRejected) {
    var promise = this
    //onResolved, onRejected用于接受上一个Promise的传参
```

```
    //它们返回的结果,用于resolve, reject方法执行下一个Promise
    return new Promise(function (resolve, reject) {
        promise.deferred.push([onResolved, onRejected, resolve,
 reject])
        promise.notify()//执行回调
    })
}
//p.then的语法糖
p.catch = function (onRejected) {
    return this.then(undefined, onRejected)
}
```

12.5.5 Promise#resolve/reject

这是框架自创的方法，用来改变Promise的状态，规范中没有。它们最终都调用Promise#notify。

```
p.resolve = function resolve(x) {
      var promise = this

      if (promise.state === PENDING) {
          if (x === promise) {
              throw new TypeError('Promise settled with itself.
')
          }

          var called = false
          try {
              var then = x && x['then']
             //如果是Promise或是thenable对象,
             //那么将执行后的结果继续传给现在这个Promise resolve /rej
cet
              if (x !== null && typeof x === 'object' && typeof
 then === 'function') {
                  then.call(x, function (x) {
                      if (!called) {
                          promise.resolve(x)
```

```
                        }
                        called = true

                    }, function (r) {
                        if (!called) {
                            promise.reject(r)
                        }
                        called = true
                    })
                    return
                }
            } catch (e) {
                if (!called) {
                    promise.reject(e)
                }
                return
            }
            promise.state = RESOLVED
            promise.value = x
            promise.notify()
        }
}

p.reject = function reject(reason) {
    var promise = this

    if (promise.state === PENDING) {
        if (reason === promise) {
            throw new TypeError('Promise settled with itself.')
        }

        promise.state = REJECTED
        promise.value = reason
        promise.notify()
    }
}
```

12.5.6 Promsie#notify

内部实现，规范并不存在，用于执行用户回调。

```
  p.notify = function notify() {
      var promise = this
      //根据Promise规范,必须异步执行存储好的回调
      nextTick(function () {
          if (promise.state !== PENDING) {//确保状态是从pending -
> resloved/rejected
              while (promise.deferred.length) {
                  var deferred = promise.deferred.shift(),
                          onResolved = deferred[0],
                          onRejected = deferred[1],
                          resolve = deferred[2],
                          reject = deferred[3]

                  try {
                      if (promise.state === RESOLVED) {
                          if (typeof onResolved === 'function')
 {
                              resolve(onResolved.call(undefined
, promise.value))
                          } else {
                              resolve(promise.value)
                          }
                      } else if (promise.state === REJECTED) {
                          if (typeof onRejected === 'function')
 {
                              resolve(onRejected.call(undefined
, promise.value))
                          } else {
                              reject(promise.value)
                          }
                      }
                  } catch (e) {
                      reject(e)
                  }
              }
          }
      })
  }
```

```
```

12.5.7 nextTick

内部实现，类似于Node.js的nextTick 或JSDeferred的next方法，用于异步执行方法。为发掘最快的异步方法，github上还有一个专门的库，叫**asap** 。

```
https://github.com/kriskowal/asap
```

这个库的作者还搞了另一个著名的Promise库，叫Q。在标准Promise没有出来时，Q、when.js与bluebird并驾齐驱，显赫一时。

```
https://github.com/kriskowal/q
https://github.com/cujojs/when
https://github.com/petkaantonov/bluebird
/*视浏览器情况采用最快的异步回调*/
var nextTick = new function () {// jshint ignore:line
    var tickImmediate = window.setImmediate
    var tickObserver = window.MutationObserver
    if (tickImmediate) {
        return tickImmediate.bind(window)
    }
```

```
    var queue = []
    function callback() {
        var n = queue.length
        for (var i = 0; i < n; i++) {
            queue[i]()
        }
        queue = queue.slice(n)
    }

    if (tickObserver) {
        var node = document.createTextNode("avalon")
        new tickObserver(callback).observe(node, {characterData
: true})
        var bool = false
        return function (fn) {
            queue.push(fn)
            bool = !bool
            node.data = bool
        }
    }

    return function (fn) {
        setTimeout(fn, 4) //标准浏览器的最小间隔数是4
    }
}
```

（1）赋值：[[Result]]=reason，[[state]]=rejected。

（2）触发[[RejectReactions]]的操作。

触发[[FulfillReactions]]和触发[[RejectReactions]]实际就是遍历数组，执行所有的

回调函数。

```
function reject(promise, reason) {
  if (promise._state !== PENDING) { return; }

  promise._state = REJECTED;
  promise._result = reason;

  asap(publish, promise);
}
```

这就完了，大概210行，实现标准Promise的所有方法。当然市面上还有其他Promise库，比如笔者刚才提到的那3个库。如果大家要在后端用Promise，建议使用bluebird。bluebird有如下优点。

（1）速度最快。

（2）api和文档完善，（对各个库支持都不错）。

（3）支持generator等未来发展趋势。

（4）github活跃。

（5）能任意同化一个对象成为一个类Promise对象。

```
var Promise = require("bluebird");
var fs = Promise.promisifyAll(require("fs"));

fs.readFileAsync("myfile.json").then(JSON.parse).then(function
(json) {
    console.log("Successful json");
}).catch(SyntaxError, function (e) {
    console.error("file contains invalid json");
}).catch(Promise.OperationalError, function (e) {
    console.error("unable to read file, because: ", e.message);
});
```

如果大家只想一个简单的Promise，可以试上面的mmPromise、ypromise、native-promise-only或者这个靠名字吃饭的es6-promise。

```
https://github.com/RubyLouvre/mmDeferred
https://github.com/yahoo/ypromise
https://github.com/getify/native-promise-only
https://github.com/stefanpenner/es6-promise
```

本节就到此为止，如果想深入理解Promise，可以看一下**native-promise-only** 的作者写的系列文

章：

《[译]深入理解 Promise 五部曲：1. 异步问题》

《[译] 深入理解 Promise 五部曲：2. 控制权转换问题》

《[译] 深入理解 Promise 五部曲：3. 可靠性问题》

《[译] 深入理解 Promise 五部曲：4. 扩展问题》

《[译] 深入理解 Promise 五部曲：5. LEGO》

扩展阅读：

```
https://zhuanlan.zhihu.com/p/23312442
https://segmentfault.com/a/1190000005051034
```

12.6 es6生成器过渡者

Generator Function（生成器函数）和Generator（生成器）是es6引入的新特性，该特性早就出现在了Python、C#等其他语言中。生成器本质上是一种特殊的迭代器。

Generator函数本意是iterator生成器，函数运行到yield时退出，并保留上下文，在下次进入时可以继续运行。

生成器函数也是一种函数，语法上仅比普通function多了个星号，即function，在其函数体内部可以使用*yield*和*yield* 关键字。

```
function* foo1() { };
function *foo2() { };
function * foo3() { };

foo1.toString(); // "function* foo1() { }"
foo2.toString(); // "function* foo2() { }"
foo3.toString(); // "function* foo3() { }"
foo1.constructor; // function GeneratorFunction() { [native cod
e] }
```

调用生成器函数会产生一个生成器（generator）。生成器拥有的最重要的方法是next()，用来迭代。

```
function* foo() { };
var bar = foo();
bar.next(); // Object {value: undefined, done: true}
```

上面第2行的语句看上去是函数调用，但这时候函数代码并没有执行；一直要等到第3行调用next方法才会执行。next方法返回一个拥有value和done两个字段的对象。

我们可以查阅co的源码，看如何判定什么是生成器与生成器函数的。

```
// https://github.com/tj/co/blob/master/index.js
function isGenerator(obj) {
  return 'function' == typeof obj.next && 'function' == typeof
obj.throw;
}

function isGeneratorFunction(obj) {
```

```
  var constructor = obj.constructor;
  if (!constructor) return false;
  if ('GeneratorFunction' === constructor.name || 'GeneratorFunction' === constructor.displayName) return true;
  return isGenerator(constructor.prototype);
}
```

生成器函数的行为与普通函数并不相同，表现为如下3点。

（1）通过new运算符或函数调用的形式调用生成器函数，均会返回一个生成器实例。

（2）通过new运算符或函数调用的形式调用生成器函数，均不会马上执行函数体的代码。

（3）必须调用生成器实例的next方法才会执行生成器函数体的代码。

```
function *enumerable(msg){
      console.log(msg)
      var msg1 = yield msg + '  after '
      console.log(msg1)
      var msg2 = yield msg1 + ' after'
      try{
        var msg3 = yield msg2 + 'after'
        console.log('ok')
      }catch(e){
```

```
        console.log(e)
      }
      console.log(msg2 + ' over')
}

// 初始化迭代器
var enumerator = enumerable('hello')
var ret = enumerator.next() // 控制台显示 hello，ret的值{value:'he
llo after',done:false}
ret =  enumerator.next('world') // 控制台显示 world，ret的值{value
:'world after',done:false}
ret = enumerator.next('game') // 控制台显示game，ret的值{value:'ga
me after',done:false}
// 抛出异常信息
ret = enumerator.throw(new Error('test'))
//控制台显示new Error('test')信息，然后显示game over。ret的值为{done:
true}
```

12.6.1 关键字yield

用于马上退出代码块并保留现场，当执行迭代器的next函数时，则能从退出点恢复现场并继续执行下去。一旦在 yield expression 处暂停，除非外部调用生成器的 next() 方法，否则生成器的代码将不能继续执行。

这使得可以对生成器的执行以及渐进式的返回值进行直接控制。

下面有两点需要注意。

（1）yield后面的表达式将作为迭代器next函数的返回值。

（2）迭代器next函数的入参将作为yield的返回值（有点像运算符）。

针对上面的例子：

```
var ret = enumerator.next()// {value:'hello after',done:false}
enumerator.next('msg1 result');//这时候msg1的值是 msg1 result;
```

12.6.2 yield*和yield的区别

yield*，一个可迭代对象，就相当于把这个可迭代对象的所有迭代值分次yield出去。

yield*，表达式本身的值就是当前可迭代对象迭代完毕时的那个返回值（也就是迭代器的迭代值的done属性为true时value属性的值）。

```
function* g1() {
   yield 2;
   yield 3;
   yield 4;
}

function* g2() {
   yield 1;
   yield* g1();
   yield 5;
}

var iterator = g2();
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: 4, done: false }
console.log(iterator.next()); // { value: 5, done: false }
console.log(iterator.next()); // { value: undefined, done: true}
```

返回值例子：

```
function* g4() {

   yield* [1, 2, 3];
   return "foo";
}
var result;
function* g5() {

result = yield* g4();
}
var iterator = g5();
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true
},
此时 g4() 返回了 { value: "foo", done: true }
```

```
console.log(result); // "foo"
```

yield的本质是一个语法糖，底层的实现方式便是CPS变换6。也就是说，yield是可以用循环和递归重新实现的，根本用不着一定在V8层面实现。但笔者认为，纯JavaScript实现的“yield”会造成大量的堆栈消耗，在性能上毫无优势可言。从性能上考虑，V8可以优化yield的编译，实现更高性能的转换。

12.6.3　异常处理

可以通过throw 抛出异常，在外层直接try catch。

```
function *foo() {

try {
    yield 2;
}
catch (err) {
    console.log( "foo caught: " + err );
}

yield; // pause

// now, throw another error
throw "Oops!";
```

```
}

function *bar() {

yield 1;
try {
    yield *foo();
}
catch (err) {
    console.log( "bar caught: " + err );
}
}
var it = bar();
it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.throw( "Uh oh!" ); // will be caught inside foo()
// foo caught: Uh oh!
it.next(); // { value:undefined, done:true } --> No error here!
// bar caught: Oops!
```

使用生成器处理串联的异步请求，比Promise更爽快。

```
//es6代码
let makeAjaxCall = (url) => {
    return new Promise((resolve, reject) => {
        // do some ajax
        resolve(result)
    })
}

makeAjaxCall('http://url1')
    .then(JSON.parse)
    .then((result) => makeAjaxCall('http://url2?q=${result.quer
y}'))
    .then(JSON.parse)
    .then((result) => makeAjaxCall('http://url3?q=${result.quer
y}'))
```

这是通过链式API将异步代码改为同步代码。

```
let makeAjaxCall = (url) => {
    // do some ajax
    iterator.next(result)
}

function* requests() {
    let result = yield makeAjaxCall('http://url1')
    result = JSON.parse(result)
    result = yield makeAjaxCall('http://url2?q=${result.query}'
)
    result = JSON.parse(result)
    result = yield makeAjaxCall('http://url3?q=${result.query}'
)
}

let iterator = requests()
iterator.next() // get everything start
```

如果改成生成器的方式，则更加直观。但是有没有办法将最后一个next调用呢？

```
let makeAjaxCall = (url) => {
    return new Promise((resolve, reject) => {
        // do some ajax
        resolve(result)
    })
}

let runGen = (gen) => {
    let it = gen()
```

```
    let continuer = (value, err) => {
        let ret

        try {
            ret = err ? it.throw(err) : it.next(value)
        } catch (e) {
            return Promise.reject(e)
        }

        if (ret.done) {
            return ret.value
        }

        return Promise
            .resolve(ret.value)
            .then(continuer)
            .catch((e) => continuer(null, e))
    }

    return continuer()
}

function* requests() {
    let result = yield makeAjaxCall('http://url1')
    result = JSON.parse(result)
    result = yield makeAjaxCall('http://url2?q=${result.query}'
)
    result = JSON.parse(result)
    result = yield makeAjaxCall('http://url3?q=${result.query}'
)
}

runGen(requests)
```

gen.next()方法获取第一个promise（见图12-9），如果迭代完毕，返回promise结束。如果迭代未完成，等待当前promise状态转化后，获取下一个

promise，然后递归调用即可。yield的作用只是实现自动promise链式调用的接口，不用人为的书写then方法。这个也是**co** 的设计思路！

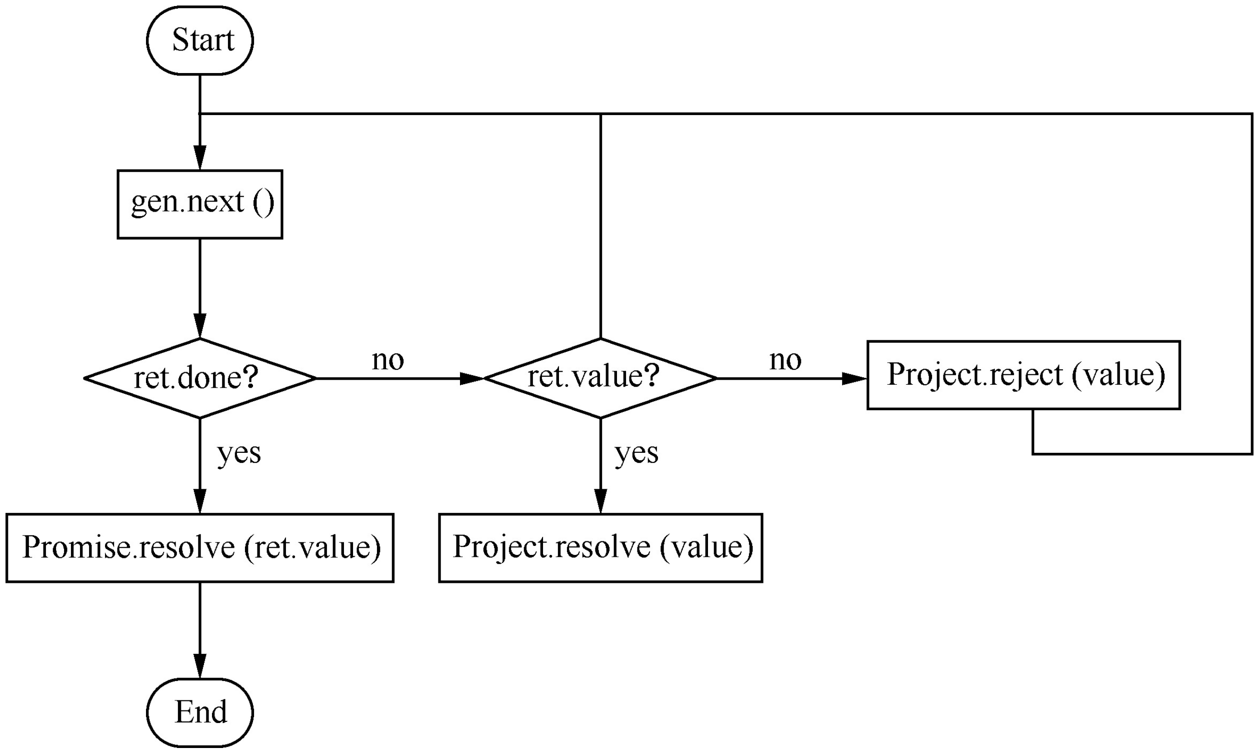


图12-9

在本书写完之前，生成器的支持情况还是相当糟，需要通过编译来使用。此外，它本来的概念也太复杂，因此原定于es7的async function很快地推出来取替它。

12.7　es7 async/await终极方案

Generator的弊病是没有执行器，它本身就不是为流程控制而生的，所以co的出现只是解决了这个问题。

可是，你不觉得奇怪吗？为什么非要加个co，才能好好的玩耍？为什么不能是直接就可以执行，并且效果和co一样的呢？

async/await就是这样被搞出来的，很多人认为它是异步操作的终极解决方案。

await的3种可能情况。

（1）await + async函数。

（2）await + Promise。

（3）await + co（co会返回Promise，这样可以

Yieldable，但难度较大，适合老手）。

前2种是比较常用的，第3种co作为promise生成器，是一种hack的办法。

我们比较一下await与生成器的用法。

```
function* gen() {
  var r1 = yield $.get('url1');
  var r2 = yield $.get('url2');
  var r3 = yield $.get('url3');
  console.log(r1, r2, r3);
}
```

然后，我们需要写一个启动器来启动这个函数。而采用async写，代码如下。

```
async function gen() {
var r1 = await $.get('url1'); var r2 = await $.get('url2');
var r3 = await $.get('url3');
console.log([r1, r2, r3].join('\n')); }
gen(); // 直接运行即可
```

直接运行，无须写生成器来运行，而代码仅仅是 * 改为 async, yield 改为 await 而已。

所以本质上讲，async就是生成的语法糖。

多任务处理有个“坑”，就是不能直接在forEach、map 之类的方法里处理，否则会报错或者得到错误的结果。

```
unction sleep(t) {
  return new Promise(resolve => setTimeout( _ => { resolve(+new
 Date) }, t))
}

async function run() {
  // 顺序
  let a = await sleep(100)
  let b = await sleep(200)

  // 并发1
  let c = await Promise.all([sleep(100), sleep(200), sleep(300)
])

  // 并发2
  let d = await Promise.all([100, 200, 300].map(t => sleep(t)))

  // 并发3
  let list = [sleep(100), sleep(200), sleep(300)]
  let e = []
  for (let fn of list) {
    e.push(await fn)
  }

  console.log(
    '',
    'a:', a, '\n',
    'b:', b, '\n',
    'c:', c, '\n',
    'd:', d, '\n',
    'e:', e, '\n'
  )
```

```
}

run()

// a: 1468317737179
// b: 1468317737384
// c: [ 1468317737485, 1468317737589, 1468317737688 ]
// d: [ 1468317737792, 1468317737890, 1468317737989 ]
// e: [ 1468317738094, 1468317738193, 1468317738293 ]
```

异常处理

Node.js里关于异常处理有一个约定，即同步代码采用try/catch，非同步代码采用error-first方式。对于async函数来说，它的await语句是同步执行的，所以最正常的流程处理是采用try/catch。

语句捕获，和generator/yield是一样的。下面的代码所展示的是通用性的做法。

```
try {
  console.log(await asyncFn());
} catch (err) {
  console.error(err);
}
```

async/await总结如下。

（1）async函数语义上非常好，让异步编程更加的同步了。

（2）async不需要启动器，它本身具备执行能力，不像Generator。

（3）async函数的异常处理采用try/catch和Promise的错误处理，非常强大。

（4）await接Promise，Promise自身就足够应对所有流程了。

（5）await释放Promise的组合能力，外加Promise的then，基本无敌。

Node.js中同样近似于async/await方式的还有asyncawait库，它不依赖generator而是依赖于node-fiber，看名字大概就是Node里的一个纤程的实现

吧。

```
https://github.com/yortus/asyncawait
```

12.8　总结

ES6的Generator本意是为了计算而设计的迭代器，但tj觉得它可以用于流程控制，于是就有了co，co的历史可以说经历了目前所有的流程控制方案，而且由于支持Generator和yield就导致yieldable。

yieldable本来是没有这个词的，因为在Generator里可以是yield关键词，而yield后面接的有5种可能，故而把这些可以yield接的方式成为yieldable，即可以yield接的。

（1）Promises。

（2）Thunks (functions)。

（3）array (parallel execution)。

（4）objects (parallel execution)。

（5）Generators and GeneratorFunctions。

这里笔者把co和promise做了简单的关联，同时区分Yieldable里的并行和顺序执行处理方式，以便大家能够更好地理解co和Yieldable，如图12-9所示。

- 顺序执行
- ◦ Promises
- ◦ Thunks
- 并行

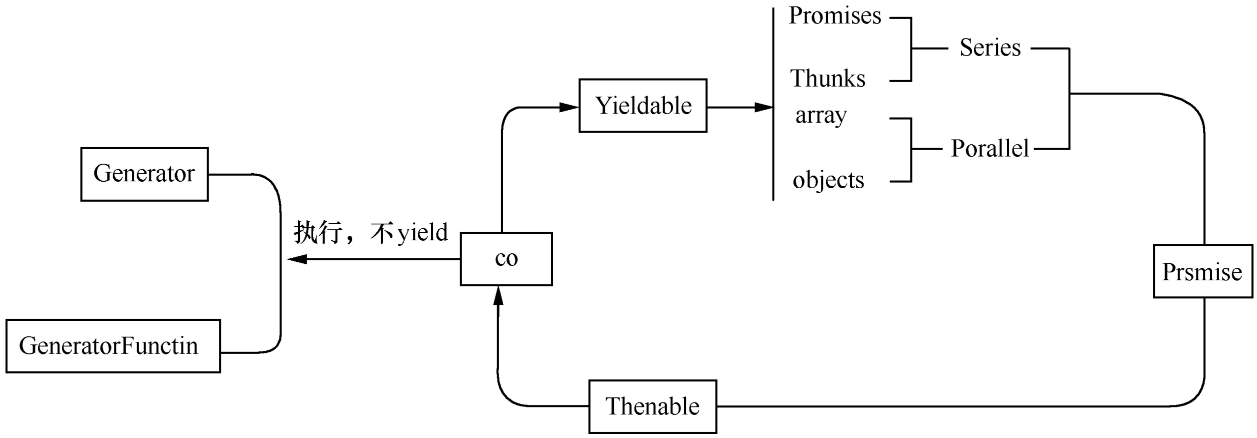


图12-10

- ◦ array
- ◦ objects

无论是哪种，它们其实都可以是Promise，而既然是Thunk对象，它们就可以thenable，而co v4.6版本的执行的返回值就是Promise，至此完成了左侧闭环。至于Generator和GeneratorFunction就要从yield和yield*讲起，在koa 1.x和2.x里有明显的应用。

最关键的是，Generator是用来计算的迭代器，它是过渡性的产物。yiedable足够强大，只是学习

成本稍高，理解起来也有些难度。

综上所述，可以得出以下结论。

（1）Async函数是趋势。

（2）Async和Generator函数里都支持promise，所以promise是必须会的。

（3）Generator和yield异常强大，不过不会成为主流，所以学会基本用法和promise就好了，没必要所有的都必须会。

（4）co作为Generator执行器是不错的，它更好的是当做Promise 包装器，通过Generator支持yieldable，最后返回Promise。

经过Callbacks, Promise, Generator, async/await，JS精英们终于造出了一种基于共享内存模型的伪线程模型，如图12-10所示。

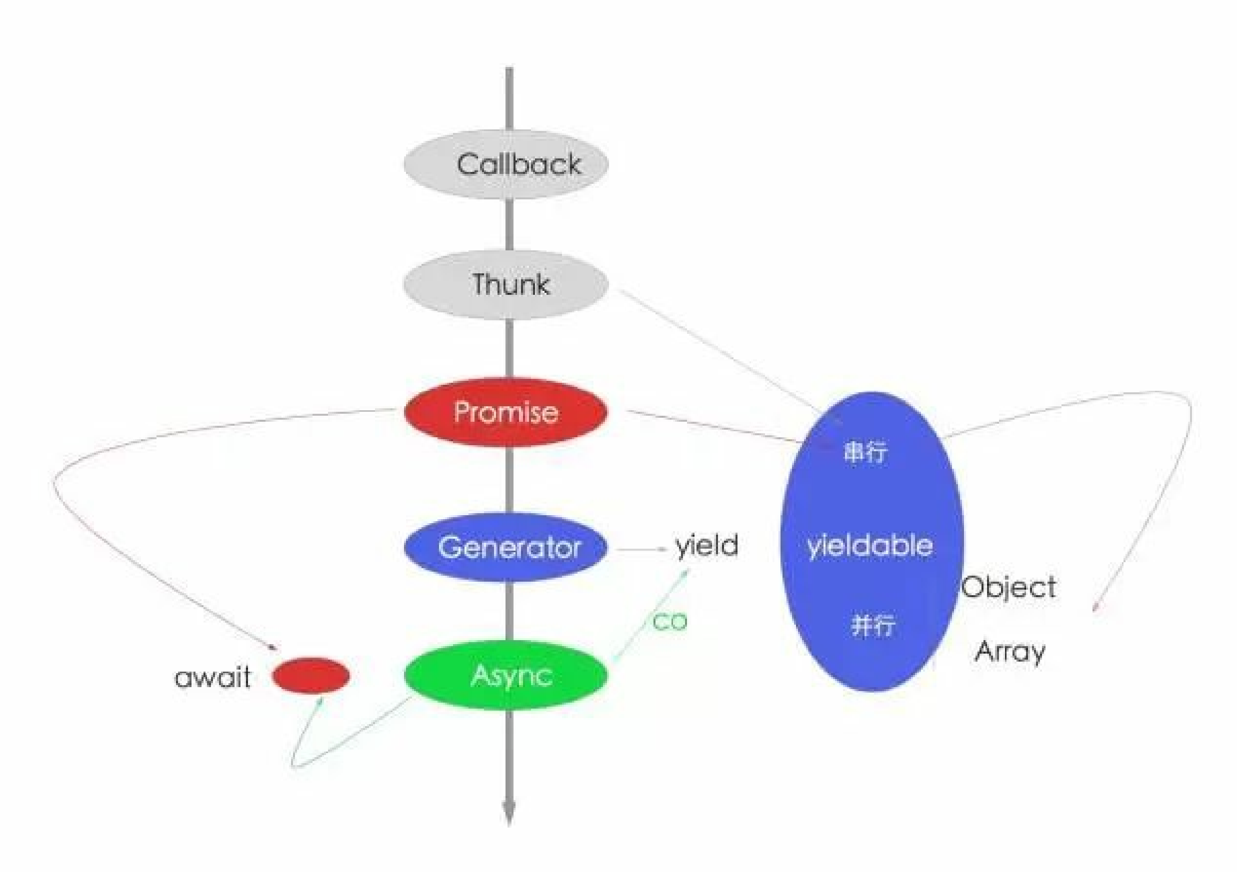
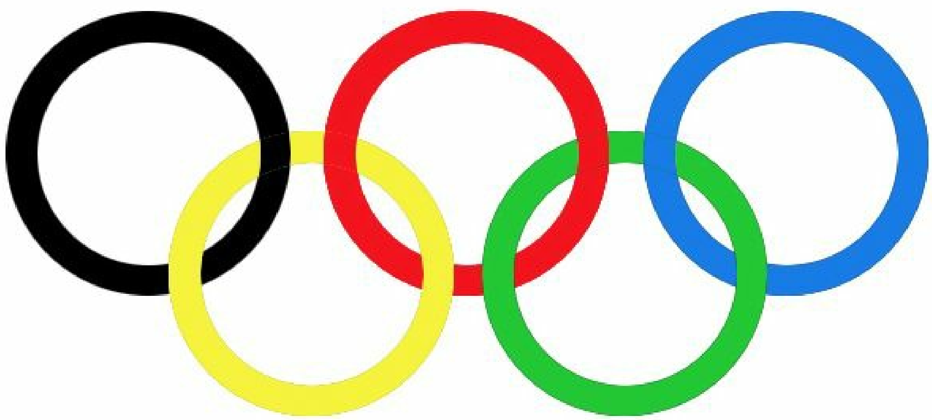

Callback
Thunk
Promise
Generator
yield
Async
co
await
串行
yieldable
并行
Object
Array

图12-11

第13章　数据交互模块



前端与后端进行交互的方式有许多种，比如直接通过地址栏请求页面、script节点加载脚本以及XMLHttpRequest对象进行更可控的数据加载。尤其是XMLHttpRequest对象，它带来的Ajax无缝刷新，直接焕发JavaScript的“第2春”，让前端向富应用发展。在许多框架与类库中，作者都是把基于script节点的JSONP作为Ajax的补充，放在一起的。本章也

不例外，还会介绍一下基于iframe的文件上传。

13.1　Ajax概览

```
var xhr = new (self.XMLHttpRequest || ActiveXObject)("Microsoft
.XMLHTTP")
xhr.onreadystatechange = function() {//先绑定事件后open
    if (this.readyState === 4 && this.status === 200) {
        var div = document.createElement("div");
        div.innerHTML = this.responseText;
        document.body.appendChild(div)
    }
}
xhr.open("POST", "/ajax", true);
//必须,用于让服务器端判断request是Ajax请求(异步)还是传统请求(同步)
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest")
xhr.send("key=val&key1=val2");
```

这是一个完整的Ajax程序，包括跨平台取得XMLHttpRequest对象，绑定事件回调，判定处理状态，发出请求，设置首部，以及在POST请求时，通过send方法发送数据。上面7个步骤每一步都有兼容性问题或易用性处理。如果是跨域请求，IE8可能比XDomainRequest更为方便。

13.2　优雅地取得XMLHttpRequest对象

Ajax的核心就是XMLhttpRequest对象，以前，人们还花大精力用iframe来模拟它，但现在可以不管了。在IE5时，微软用一个ActiveXObject对象来加载数据，并且在数据返回时不会导致地址栏跳转和页面刷新。除此以外，微软的ActiveXObject还可以做许多事情，比如创建一个HTML页面或XML文档，解析XSLT，外挂Windows的日历，拖动条什么。因此要知道这个ActiveXObject是干什么用，必须传参。像XMLhttpRequest这样重要的对象，在其发展过程中，一直面临外界的剧烈竞争，因此它也不断升级，这些版本号与名字连在一起，组成参数传入ActiveXObject才能正确生成XMLhttpRequest。但它们之间没有什么规则，Msxml2.XMLHTTP.6.0、Msxml2.XMLHTTP.5.0、Msxml2.XMLHTTP.4.0、Msxml2.XMLHTTP.3.0、

Msxml2.XMLHTTP、Microsoft.XMLHTTP，逐个试验（这取决于用户有没有打补丁与操作系统自带的版本）。

毋庸置疑，越新的版本功能越多，因此我们在试验时，把最新的版本放在前面。可能微软最后也发觉这用户体验太差了，也学标准浏览器那样，直接提供一个XMLHttpRequest对象给你new，然后内部总是对应当前可用的最新版本。

IE7这个决定是明智的，但IE7在IE的发展史，是个半成品，这个XMLHttpRequest对象也一样不好用。它不支持本地file协议，会出现拒绝访问，需要倒退到ActiveXObject对象。如果服务器不发送任何header来禁止浏览器进行缓存的话，IE7的XMLHttpRequest对象可能会缓存和重用对GET请求的响应。使用以下的解决方式仍然会出现该问题：使用POST方法、使用随机的查询串（加时间戳

等）、配置并使服务器发送某些缓存的指令。注意：单独使用一个非随机的查询串并不能阻止浏览器进行缓存。此外，IE7的XMLHTTPRequest对象与标准浏览器的出入还是很大，人家有prototype，有onbort、onload、onerror方法。它还不是一个JavaScript对象，是MSXML2.XMLHTTP.3.0的外壳。

如果我们查看一下各大类库的源码，发现它们并没有全部逐一试验。

```
//*******************jQuery1.4a2*****************
xhr: function(){
  return window.ActiveXObject ?
    new ActiveXObject("Microsoft.XMLHTTP") :
    new XMLHttpRequest();
},
//*******************mootools1.2.4*****************
Browser.Request = function(){
  return $try(function(){
    return new XMLHttpRequest();
  }, function(){
    return new ActiveXObject('MSXML2.XMLHTTP');
  }, function(){
    return new ActiveXObject('Microsoft.XMLHTTP');
  });
};
//*********************prototype1.61rc2******************
getTransport: function() {
  return Try.these(
```

```
  function() {return new XMLHttpRequest()},
  function() {return new ActiveXObject('Msxml2.XMLHTTP')},
  function() {return new ActiveXObject('Microsoft.XMLHTTP')}
) || false;
},
```

Microsoft.XMLHTTP是最早的版本，Msxml2.XMLHTTP是IE6的，但打了补丁后自然有Msxml2.XMLHTTP.3.0和Msxml2.XMLHTTP.4.0，Msxml2.XMLHTTP.5.0不是给浏览器使用的，属于旁支，存在一定的兼容问题。其中5.0是为office所开发的，甚至带有一些特性是后来的6.0所没有的（如xml数字加密）。

MSXML 4.0 is a separate download that was released by Microsoft in October 2001. The latest or current service pack release of MSXML 4.0 is available through the Microsoft Web site. MSXML 4.0 must be installed separately and is not currently included with other Microsoft products. MSXML 4.0

installs side-by-side with earlier versions of MSXML without affecting any existing functionality.

MSXML 5.0 for Microsoft Office Applications is only available with current versions of Microsoft Office. MSXML 5.0 for Microsoft Office Applications installs side-by-side with earlier versions of MSXML without affecting any existing functionality.

Msxml2.XMLHTTP.6.0是已知ActiveXObject系的XMLHttpRequest对象的最高版本。可能大家在网上还看到Msxml2.XMLHTTP.7.0，很遗憾，那只是以讹传讹，笔者用IE8测试并不存在这个版本。

根据IE blog的建议，应该仅使用6.0和3.0，不要使用旧的microsoft.xmlhttp，那它也应该出现在上面的数组中。此外还有一个MSXML2.XMLHTTP.2.6，但不知为什么，总之，如果你的浏览器打了某些升级补丁，new

ActiveXObject（“Msxml2.XMLHTTP”）调用的是2.6或3.0版本，非常混乱。此后的版本，才正确对应它的版本号。因此上面没有列举Msxml2.XMLHTTP.2.6与Msxml2. XMLHTTP.3.0的必要，都给Msxml2.XMLHTTP代表了。

因此我们要检测的ActiveXObject的ProgID收窄为Msxml2.XMLHTTP.6.0、Msxml2. XMLHTTP.3.0、Msxml2.XMLHTTP与Microsoft.XMLHTTP4个。

```
function xhr() {
  if(!xhr.cache){
    var fns = [
      function () { return new XMLHttpRequest(); },
      function () { return new ActiveXObject('Msxml2.XMLHTTP');
 },
      function () { return new ActiveXObject('Microsoft.XMLHTTP
'); },
    ];
    for (var i = 0,n=fns.length; i < n; i++) {
      try {
        fns[i]();
        xhr.cache = fns[i];
        break;
      }catch(e){}
    }
    return xhr.cache();
  }else{
    return xhr.cache();
```

```
  }
}
var xhrObject = xhr();//调用
alert(xhrObject) //[object XMLHttpRequest]
```

凭心而论，上面的方法已经很高效了，只判定了一次，然后缓存生成方式。但有没有更优雅的设计呢？

因为即便我们缓存了生成方式，每次还要判断一下xhr.cache的值。这时用惰性函数处理，这是一种覆写自身的模式。

```
var xhr =  function() {
  var fns = [
    function () { return new XMLHttpRequest(); },
    function () { return new ActiveXObject('Msxml2.XMLHTTP'); }
,
    function () { return new ActiveXObject('Microsoft.XMLHTTP')
; },
  ];
  for (var i = 0,n=fns.length; i < n; i++) {
    try {
      fns[i]();
      xhr = fns[i];//注意这里，覆写自身
      break;
    }catch(e){}
  }
  return xhr()
}
```

我们再认真思考一下，既然我们是写框架，那么这些检测其实是放在IIFE里面，因此基本不用覆写，检测好哪个可用，就把它加到命名空间上就好了。最后的版本就出来了，使用new Function、eval，反正只用一次，耗不了多少性能。

```
window.$ = window.$ = {}
var s = ["XMLHttpRequest", "ActiveXObject('Msxml2.XMLHTTP.6.0')
",
    "ActiveXObject('Msxml2.XMLHTTP.3.0')", "ActiveXObject('Msxm
l2.XMLHTTP')"];
if (!"1"[0]) {//判定IE67
    s[0] = location.protocol === "file:" ? "!" : s[0];
}
for (var i = 0, axo; axo = s[i++]; ) {
    try {
        if (eval("new " + axo)) {
            $.xhr = new Function("return new " + axo);
            break;
        }
    } catch (e) {
    }
}
```

13.3　XMLHttpRequest对象的事件绑定与状态维护

最早的XMLHTTPRequest对象只有onreadystatechange方法，然后被标准浏览器抄来，在易用性上进行增强。首先将XMLHTTPRequest改造成一个事件派发者（EventDispatcher）。很早以前，浏览器只有3种原生的事件派发者，window对象、文档对象与元素节点。既然是事件派发者，就有addEventListener、removeEventListener、dispatchEvent等多投事件API，IE到9.0才支持W3C那套，因此它的XMLHTTPRequest也在IE9才有addEventListener这些API。

早些年，W3C的大旗一直是Firefox在扛，许多API都是Firefox搞出来的，在漫长的Firefox3.X时代，onload、onerror、onabort、onprogress这4个方法首先被搞出来。其他标准浏览器只是跟进。微软

在IE8才为XDomainRequest添加onerror、onload、onprogress、ontimeout这几个事件，或许觉得过期事件非常有用，IE8为XMLHTTPRequest多加个ontimeout事件，IE9又加个onabort事件。

Firefox在3.6开始支持用XMLHttpRequest上传二进制文件，为此搞出FileReader对象，此对象所支持的事件类型与后来的XMLHTTPRequest2的一模一样（onabort、onerror、onload、onloadend、onloadstart、onprogress）。其中onloadend与jQuery ajax的complete回调一模一样，无论成功与错误都会触发，用于收尾工作非常适合。IE10终于把这些事件一口气实现了，如表13-1所示。

表**13-1**

事件	描述
loadstart	在请求开始时触发
progress	在请求发送或接收数据期间，在服务器指定的时间间隔触发

abort	在请求被取消时触发，例如，在调用abort()方法时
error	在请求失败时触发
load	在请求成功完成时触发
timeout	在指定的时间段已经结束时触发
loadend	在请求完成时触发，无论请求是成功还是失败
readystatechange	在XHR对象的readyState值发生改变时触发

从实现角度来看，由于IE6～IE8的XMLHTTPRequest无法进展原型，我们需要用包裹的方式创建一个伪XMLHTTPRequest对象，在它里面操作原生对象。对于事件绑定，为了对同一种事件绑定多个回调，我们需要继承一个自定义事件对象，换言之，一个观察者模式的东西。loadstart就是在一开头执行，没什么难度，成功与失败我们可以判定status状态码，ontimeout可以用setTimeout实现，onabort就是一个开关，loadend就是在回调里肯

定会执行的方法。最麻烦是onprogress，在标准浏览器中我们可以通过事件对象的loaded与total属性轻易计算得进度，IE在readyState==3时Content-Length的值并不可靠。

至于请求是成功还是失败，IE就要在readystatechange回调中查看status值与转换目标类型是否成功。

2xx状态与表示从缓存中直接取出的304可以看是成功，但浏览器还是有一些例外情况需要我们注意。IE（非原生的XHR对象）中会将204设置为1223，Opera会在取得204时将status设置为0，而Safari 3之前的版本会将status设置为undefined。最终验证请求是否成功的代码将会是：

```
var ok = ( xhr.status >= 200 && xhr.status < 300 ) ||
 xhr.status === 304 || xhr.status === 1223 || xhr.status === 0
```

IE的1223请求算是一个著名的bug，在各大类库的bugstack中都有介绍。

```
XMLHTTPRequest implementation in MSXML HTTP (at least in IE 8.0
 on Windows XP SP3+) does not handle HTTP responses with status
 code 204 (No Content) properly; the `status' property has the
value 1223.
dojo - http://trac.dojotoolkit.org/ticket/2418
prototype - https://prototype.lighthouseapp.com/projects/8886/t
ickets/129-ie-mangles-http-response- status-code-204-to-1223
YUI - http://developer.yahoo.com/yui/docs/connection.js.html (h
andleTransactionResponse)
JQuery - http://bugs.jquery.com/ticket/1450
ExtJS - http://www.sencha.com/forum/showthread.php?85908-FIXED-
732-Ext-doesn-t-normalize- IE-s-crazy-HTTP-status-code-1223
```

IE下甚至会返回5位数的状态码，下面是WinInet 错误代码。

```
http://support.microsoft.com/kb/193625
```

另外，Firefox在本地使用XMLHttpRequest时，成功时status为0。由于很少在本地发请求，因此主流框架没有对它加以处理。这个当作常识，自己留意一下吧。

13.4　发送请求与数据

XMLHTTPRequest对象发送请求是使用open方法，在这之前请先绑定好各种事件回调。

语法如下：

```
open(method, url, async, username, password)
```

method参数是用于请求的HTTP方法。值包括GET、POST、PUT、DELETE 和 HEAD。有的浏览器还允许你自定义method，不过要求全是大写，比如IE6、Firefox3～Firefox19、Chrome、Opera。

url 参数是请求的主体。大多数浏览器实施了一个同源安全策略，并且要求这个URL与包含脚本的文本具有相同的主机名和端口。在GET请求，我们需要将参数转换成querystring的形式放在问号后

面。

async参数指示请求使用应该异步地执行。如果这个参数是false，请求是同步的，后续对send()的调用将阻塞，直到响应完全接收。如果这个参数是true或省略，请求是异步的，且通常需要一个onreadystatechange 事件句柄。

username和password参数是可选的，为url所需的授权提供认证资格。如果指定了，它们会覆盖url自己指定的任何资格。

发送数据要用send方法，网上通常教我们在POST请求时发送querystring。后来增加了FormData、ArrayBuffer、Blob、Document这几种数据类型。FormData是一个不透明的对象，无法序列化，但能简化人工提交数据的过程。以前，我们点击按钮提交表单，浏览器会自动将这个表单的所有disabled为false的input、texteara、select、button元素

的name与value抽取出来，变成一个querystring。当我们用Ajax提交时，这个过程就成为人工的。jQuery把它抽象成一个serialize方法，代码量不是少数。而FormData直接可以new一个实例出来，我们只需遍历表单元素，用append方法传入其name与value就行了。

```
var formdata = new FormData();
formdata.append("name", "司徒正美");
formdata.append("blog", "http://www.cnblogs.com/rubylouvre/");
```

更简捷的方式，它本来就可以用getFormData生成，并得到此表单的所有数据。

```
var formobj =  document.getElementById("form");
var formdata = formobj.getFormData()
```

它也可以用传参方法填充内容。

```
var formobj =  document.getElementById("form");
var formdata = new FormData(formobj);
```

最后就是提交。

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://ajaxpath");
xhr.send(formData);
```

如果是document，就自己生成一个XML对象发上去吧，但我们没有好的手段辨识浏览器是否支持send(document)。对于其他新数据，一般来说，只要它们的构造器是出现在全局作用域下的，浏览器就已经同步好send方法了。

在标准浏览器支持二进制的过程中，无节操地实现了各种各样的对象。有的只是昙花一现，很快被废弃掉，如BlobBuilder。剩下的还有Blob、File、FileReader、FileWriter、BlobURL及庞大的TypedArray家族。

下面是发送ArrayBuffer与Blob的例子。

```
var myArray = new ArrayBuffer(512);
```

```
var longInt8View = new Uint8Array(myArray);
for (var i = 0; i < longInt8View.length; i++) {
    longInt8View[i] = i % 255;
}
var xhr = new XMLHttpRequest;
xhr.open("POST", url, false);
xhr.send(myArray);
var xhr = new XMLHttpRequest();
xhr.open("POST", url, true);
var blob = new Blob(['abc123'], {type: 'text/plain'});
xhr.send(blob);
```

Firefox很早以前还实现了一个私有的sendAsBinary，可直接发送二进制数据。Chrome可以用以下方法模拟。

```
XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
    var bb = new WebKitBlobBuilder();
    var data = new ArrayBuffer(1);
    var ui8a = new Uint8Array(data, 0);
    for (var i in datastr) {
        if (datastr.hasOwnProperty(i)) {
            var chr = datastr[i];
            var charcode = chr.charCodeAt(0)
            var lowbyte = (charcode & 0xff)
            ui8a[0] = lowbyte;
            bb.append(data);
        }
    }
    var blob = bb.getBlob();
    this.send(blob);
}
```

13.5　接收数据

早期XMLHTTPRequest对象拥有两种接收数据的属性，responseText对应解码后的字符串（默认解码为utf-8），responseXML对应一个XML文档，IE还支持第3种，responseBody对应未解码的二进制数据。JSON传输格式兴起后，我们会对responseText进行加工，用JSON.parse得到JSON数据。至于后端返回什么类型的数据，在项目开发过程，这个有对应文档看，如果比较悲催，我们还可以通过getResponseHeader(“Content-Type”)得知。

随着浏览器着手对二进制的支持，它新增的responseType和response属性，告知浏览器我们希望返回什么格式的数据。

responseType，在发送请求前，根据您的数据需要，将xhr.responseType设置为“text”

“arraybuffer”“blob”或“document”。请注意，设置（或忽略）xhr.responseType =，会默认将响应设为“text”。

response成功发送请求后，xhr的响应属性会包含DOMString、ArrayBuffer、Blob或Document 形式（具体取决于responseType的设置）的请求数据。

方法一：

```
适用范围：Chrome8～Chrome24、Firefox6～Firefox18、IE10。
var BlobBuilder = window.MozBlobBuilder || window.WebKitBlobBui
lder || window.MSBlob Builder || window.BlobBuilder
if (!BlobBuilder) {
    console.log("BlobBuilder 已被废弃")
}
var xhr = new XMLHttpRequest();
var img = document.getElementById("img")
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.responseType = 'arraybuffer';
xhr.onload = function(e) {
    if (this.status === 200) {
        var bb = new BlobBuilder();
        bb.append(this.response);
        var blob = bb.getBlob('image/jpeg');
        img.src = blob;
    }
};
xhr.send();
```

方法二：

```
//适用范围：Chrome19+、Firefox6、IE10、Opera12、Safari5
var xhr = new XMLHttpRequest();
var img = document.getElementById("img");
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.responseType = 'blob';
//https://developer.mozilla.org/zh-CN/docs/DOM/window.URL.creat
eObjectURL
window.URL = window.URL || window.webkitURL;
img.addEventListener("DOMNodeRemoved", function() {
    window.URL.revokeObjectURL(img.src);
});
xhr.onload = function(e) {
    if (this.status === 200) {
        var blob = this.response;
        img.src = window.URL.createObjectURL(blob);
    }
}
xhr.send();
```

方法三：

```
//适用范围：Chrome10+、Firefox6、IE10、Opera11.60
var xhr = new XMLHttpRequest();
xhr.open('POST', '/path/to/image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.responseType = 'arraybuffer';
xhr.onload = function(e) {
    if (this.status == 200) {
        var uInt8Array = new Uint8Array(this.response);
        var i = uInt8Array.length;
        var binaryString = new Array(i);
        while (i--) {
            binaryString[i] = String.fromCharCode(uInt8Array[i]
);
            }
```

```
        var data = binaryString.join('');
        var base64 = window.btoa(data);
        document.getElementById("img").src = "data:image/jpeg;base64," + base64;
        }
    };
xhr.send();
```

方法四：

```
//适用范围：支持overrideMimeType与btoa的浏览器
var xhr = new XMLHttpRequest();
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
//重写了默认的MIME类型，强制浏览器将该响应当成纯文本文件来对待，使用一个用户自定义的字符集
//这样就是告诉了浏览器，不要去解析数据,直接返回未处理过的字节码
xhr.overrideMimeType('text/plain; charset=x-user-defined')
xhr.onload = function(e) {
    if (this.status == 200) {
        var responseText = xhr.responseText;
        var responseTextLen = responseText.length;
        var binary = ''
        for (var i = 0; i < responseTextLen; i += 1) {
            //扔掉的高位字节(f7)
            binary += String.fromCharCode(responseText.charCodeAt(i) & 0xff);
        }
        var base64 = window.btoa(binary);
        document.getElementById("img").src = "data:image/jpeg;base64," + base64;
    }
};
xhr.send();
```

btoa其实就是一个encodeBase64方法。如果浏

览器不支持，可以自己实现一个，让适用范围更广。

```
(function() {
var chars =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
+/'.split('');
    window.bota = window.btoa || function(str) {
        if (/[^\u0000-\u00ff]/.test(str)) {
            throw new Error('encodeBase64 : INVALID_CHARACTER_E
RR');
        }
        var hash = chars;
        var i = 2;
        var len = str.length;
        var output = [];
        var link;
        var mod = len % 3;
        len = len - mod;
        for (; i < len; i += 3) {
            output.push(
                    hash[(link = str.charCodeAt(i - 2)) >> 2],
                    hash[((link & 3) << 4 | ((link = str.charCo
deAt(i - 1)) >> 4))],
                    hash[((link & 15) << 2 | ((link = str.charC
odeAt(i)) >> 6))],
                    hash[(link & 63)]
                    );
        }
        if (mod) {
            output.push(
                    hash[(link = str.charCodeAt(i - 2)) >> 2],
                    hash[((link & 3) << 4 | ((link = str.charCo
deAt(i - 1)) >> 4))],
                    link ? hash[(link & 15) << 2] + '=' : '=='
                    );
        }
        return output.join('');
    }
})()
```

下面讲解如何对付旧版本IE。IE支持两种脚本语言，JavaScript与VBScript。JavaScript不能做的事，VBScript可以殿后。整个思路是参照方式4，搞一个类似的Uint8Array的东西出来。

你可以使用以下两种方式创建一个VBS的转换函数。

```
(function() {
    function vbs() {
        /*
Function BinaryToArray(binary)
    Dim length,array()
    length = LenB(binary) - 1
    ReDim array(length)
    For i = 0 To length
        array(i) = AscB(MidB(binary, i + 1, 1))
    Next
    BinaryToArray = array
End Function
         */
    }
    var str = vbs.toString();
    execScript(str.slice(str.indexOf("*") + 1, str.lastIndexOf(
"*")), "VBScript");
})();
(function() {
    var str =
            'Function BinaryToArray(binary)\r\n\
                 Dim oDic\r\n\
                 Set oDic = CreateObject("scripting.dictionary"
)\r\n\
```

```
                 length = LenB(binary) - 1\r\n\
                 For i = 1 To length\r\n\
                     oDic.add i, AscB(MidB(binary, i, 1))\r\n\
                 Next\r\n\
                 BinaryToArray = oDic.Items\r\n\
              End Function'
    execScript(str, "VBScript");
})();
```

第一种是使用多行注释实现heredocument，但不防压缩。第二种为了保持格式漂亮加入了许多空白，注意不能去掉后面的换行符，否则解释错误。通过BinaryToArray方法得到一个数字数组，但它是一个VBScript对象，我们需要把它放到VBArray构造器内，转换为真正的JavaScript数组。剩下的处理就是一样的了。

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.onload = function(e) {
    if (this.status === 200) {
        var byteArray = new VBArray(BinaryToArray(this.response
Body)).toArray();
        var n = byteArray.length;
        var binary = '';
        for (var i = 0; i < n; i++) {
            //扔掉的高位字节(f7)
            binary += String.fromCharCode(byteArray[i] & 0xff);
        }
        var base64 = window.btoa(binary);
```

```
        document.getElementById("img").src = "data:image/jpeg;base64," + base64;
    }
};
xhr.send();
```

13.6　上传文件

这个与前面提到的传送数据有点相似，不过它与input[type=file]结合得更紧密。

假设页面上有一个ID为upload的上传域与一个ID为progress的用于显示进度的SPAN元素，那么使用XMLHTTPRequest2来上传文件是这样实现的：

```
window.addEventListener("load", function() {
    var el = document.querySelector('#file');
    var progress = document.querySelector('#progress');
    el.addEventListener('change', function() {
        var file = this.files[0];
        if (file) {
            var xhr = new XMLHttpRequest();
            xhr.upload.addEventListener('progress', function(e)
 {
                //处理兼并性问题，不同版本其名字不一样
                var done = e.position || e.loaded, total = e.to
talSize || e.total;
                progress.innerHTML = (Math.floor(done / total *
 1000) / 10) + "%";
            });
            xhr.addEventListener('load', function() {
                progress.innerHTML = "上传成功";
            });
            xhr.open('PUT', '/upload', true);
            xhr.setRequestHeader('X-Requested-With', 'XMLHttpRe
quest');
            xhr.setRequestHeader('X-File-Name', encodeURICompon
ent(file.fileName || file.name));
```

```
            xhr.setRequestHeader('Content-Type', 'application/o
ctet-stream');
            xhr.send(file);
        }
    })
});
```

如果文件很大，我们利用文件对象的slice方法切割一下，分块上传。

```
window.addEventListener("load", function() {
    var el = document.querySelector('#file');
    var progressBar = document.querySelector('#progress');
    function upload(name, index, file, total) {
        var xhr = new XMLHttpRequest();
        xhr.addEventListener('load', function() {
            progressBar.innerHTML = "100% 上传成功";
        });
        xhr.open('PUT', '/upload', true);
        xhr.setRequestHeader('X-Requested-With', 'XMLHttpReques
t');
        xhr.setRequestHeader('X-File-Index', index);
        xhr.setRequestHeader('X-File-total', total);
        xhr.setRequestHeader('X-File-Name', encodeURIComponent(
name));
        xhr.setRequestHeader('Content-Type', 'application/octet
-stream');
        xhr.send(file);
    }
    el.addEventListener('change', function() {
        var blob = this.files[0];
        var meanSize = 512 * 1024; // 1MB chunk sizes.
        var totleSize = blob.size;
        var start = 0, end = 0;
        var i = 0;
        var name = blob.fileName || blob.name;
        var total = Math.ceil(totleSize / meanSize)
        while (start < totleSize) {
```

```
            if ('mozSlice' in blob) {
                var chunk = blob.mozSlice(start, end);
            } else if ("webkitSlice" in blob) {
                chunk = blob.webkitSlice(start, end);
            } else {
                chunk = blob.slice(start, end);
            }
            upload(name, i, chunk, total);
            i++
            start = end;
            end = start + meanSize;
        }
    });
})
```

至于旧式IE，那就没有办法了，需要用到iframe或flash。不过专业的上传组件都是混用多种上传手段，非常庞大，笔者这里不就展开了。这里稍微列出笔者所知道的成熟方案，如图13-1所示。

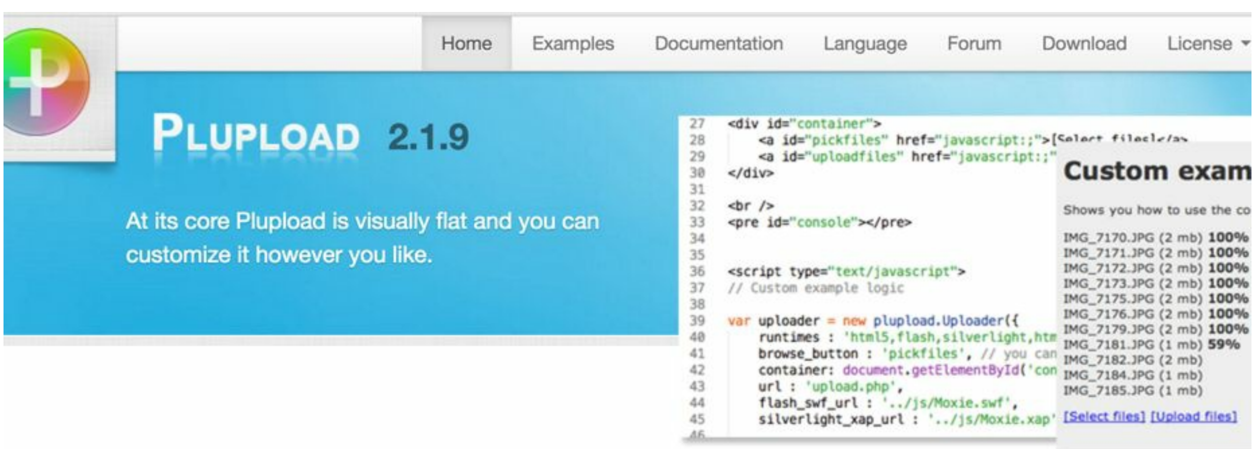


图13-1

13.7　jQuery.ajax

Ajax的底层实现都是浏览器提供的，所以任何基于API上面的框架或者库，都只是说对于功能的灵活与兼容维护性做出最优的扩展。

ajax请求的流程：

（1）通过new XMLHttpRequest或其他的形式（指IE）生成Ajax的对象xhr。

（2）通过xhr.open(type, url, async, username, password)的形式建立一个连接。

（3）通过setRequestHeader设定xhr的请求头部（request header）。

（4）通过send(data)请求服务器端的数据。

（5）执行在xhr上注册的onreadystatechange回

调处理返回数据。

这几步之中，我们开发者可能会遇到如下的问题。

（1）跨域处理。

（2）将返回的原始数据转换为想要的格式。

（3）Ajax乱码问题。

（4）页面缓存。

（5）HTTP状态的纠正与维护。

（6）不同平台兼容。

jQuery主要就是解决上面这问题，之后就在这个基础之上进行扩展jQuery 2.0.3版的Ajax部分，源码大概有1200多行，主要针对Ajax的操作进行了一些扩展，使之更加灵活。

jQuery在1.5中对Ajax模块的重写，增加了几个新的概念

Ajax模块提供了3个新的方法用于管理、扩展Ajax请求，分别是。

（1）前置过滤器jQuery. ajaxPrefilter。

（2）请求分发器jQuery. ajaxTransport。

（3）类型转换器ajaxConvert。

除此之后还重写了整个异步队列处理，加入了deferred，可以将任务完成的处理方式与任务本身解耦合。使用deferreds对象，多个回调函数可以被绑定在任务完成时执行，甚至可以在任务完成后绑定这些回调函数。这些任务可以是异步的，也可以是同步的。

1. 链式反馈done与fail

```
$.ajax({
    url: "script.php",
    type: "POST",
    data: {
        id: menuId
    },
    dataType: "html"
}).done(function(msg) {
    $("#log").html(msg);
}).fail(function(jqXHR, textStatus) {
    alert("Request failed: " + textStatus);
});
```

2. 分离异步与同步处理

```
var ajax = $.ajax({
    url: "script.php",
    type: "POST",
    data: {
        id: menuId
    },
    dataType: "html"
}).fail(function(jqXHR, textStatus) {
    alert("Request failed: " + textStatus);
});

//同步还在执行代码，这个函数有可能在Ajax结束前调用
dosomething()
//同步还在执行代码，这个函数有可能在Ajax结束前调用
dosomething()

//异步还在等在成功响应
ajax.done(function(msg) {
    $("#log").html(msg);
})
```

不再被限制到只有一个成功，失败或者完成的回调函数了。相反这些随时被添加的回调函数被放置在一个先进先出的队列中。

3．同时执行多个Ajax请求

```
function ajax1() {
    return $.get('1.htm');
}

function ajax2() {
    return $.get('2.htm');
}

$.when(ajax1(), ajax2())
    .then(function() {
       //成功
    })
    .fail(function() {
       //失败
    });
```

显而易见，deferred对象就是jQuery的回调函数解决方案，它解决了如何处理耗时操作的问题，对那些操作提供了更好的控制，以及统一的编程接口。

jqXHR对象

从jQuery 1.5开始，$.ajax()返回XMLHttpRequest（jqXHR）对象，该对象是浏览器的原生的XMLHttpRequest对象的一个超集。用于屏蔽各种不同传输对象的差异，比如说，旧式IE是使用ActiveXObject，如果要跨域有些jQuery插件会替换为XDomainRequest，标准浏览器使用XMLHttpRequest，但XMLHttpRequest分为第一代与第二代，当你要JSONP时，jQuery会使用script标签来发送请求，但你要上传时，jQuery.form会用iframe。因此必须包一层，无法最终用什么东西发送请求，在回调里面的this总是为一个类似XMLHttpRequest的对象（jqXHR对象）。

一个jqXHR对象拥有以下属性与方法，如图13-2所示。

- readyState。
- status。

- statusText。
- responseXML/responseText。
- setRequestHeader(name, value)。
- getAllResponseHeaders()。
- getResponseHeader()。
- abort()。

jQuery.Deferred出来后，通过Deferred的promise方法，再为它添加更多添加链式回调的方法，如图13-3所示。

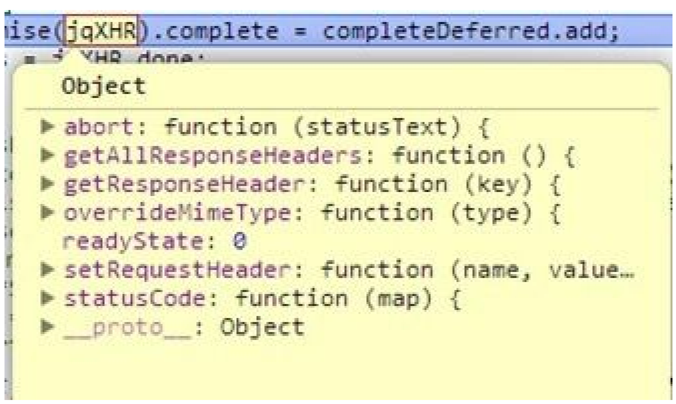


图13-2

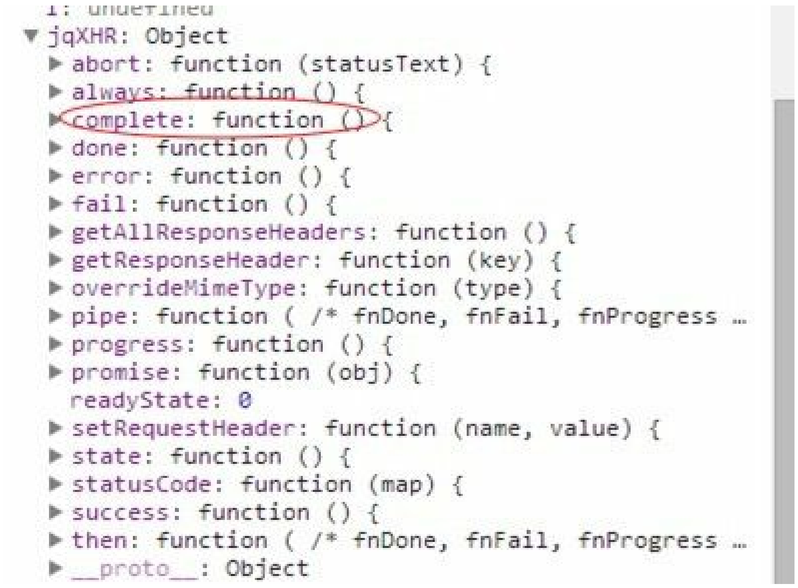


图13-3

```
deferred = jQuery.Deferred(),
deferred.promise( jqXHR );
```

此外，jQuery在jQXHR短暂的生命周期中也能触发各种事件，我们可以订阅这些事件并在其中处理我们的逻辑。在jQuery中有两种Ajax事件：局部事件和全局事件。

局部事件（回调函数），在$.ajax()方法的options参数中声明，可以用来设置请求数据和获取、处理响应数据。

（1）beforeSend该函数可在发送请求前修改XMLHttpRequest对象，如添加自定义HTTP头。

（2）dataFilter在请求成功之后调用。若状态码为304（未修改）则不触发此回调。

（3）success请求成功时触发。

（4）error请求失败时调用此函数。

（5）complete请求完成后回调函数（请求成功或失败之后均调用）。

全局事件，每次Ajax请求都会触发，它会向DOM中的所有元素广播，你只需为DOM中任意元素bind好全局事件即会触发（若绑定多次，则会依次触发为事件注册的回调函数）。

（1）ajaxStart开始新的Ajax请求，并且此时jQuery对象上没有其他Ajax请求正在进行。

（2）ajaxSend当一个Ajax请求开始时触发。

（3）ajaxSuccess全局的请求成功。

（4）ajaxError全局的发生错误时触发。

（5）ajaxComplete全局的请求完成时触发。

（6）ajaxStop当jQuery对象上正在进行Ajax请求都结束时触发。

此外，jQuery还提供了几个提交数据时经常用的数据序列化函数。

`serialize` 用于序列化一组表单元素，将表单内容编码为用于提交的字符串。

`serializeArray` 用于序列化一组表单元素，将表单内容编码为一个JavaScript数组。

`param` 将一个JS数组或对象序列化为字符串

值，以便用于URL查询字符串或Ajax请求。

总结，jQuery.ajax是目前世界上最好的Ajax程序了，支持Promise，功能齐全，扩展性强。当然它也有一个缺点，就是与jQuery绑在一起，有时为了简化Ajax编码就引入整个jQuery未免有些浪费，则可以试试这些专业库：

1．成功的Ajax请求的事件流（见图13-4）

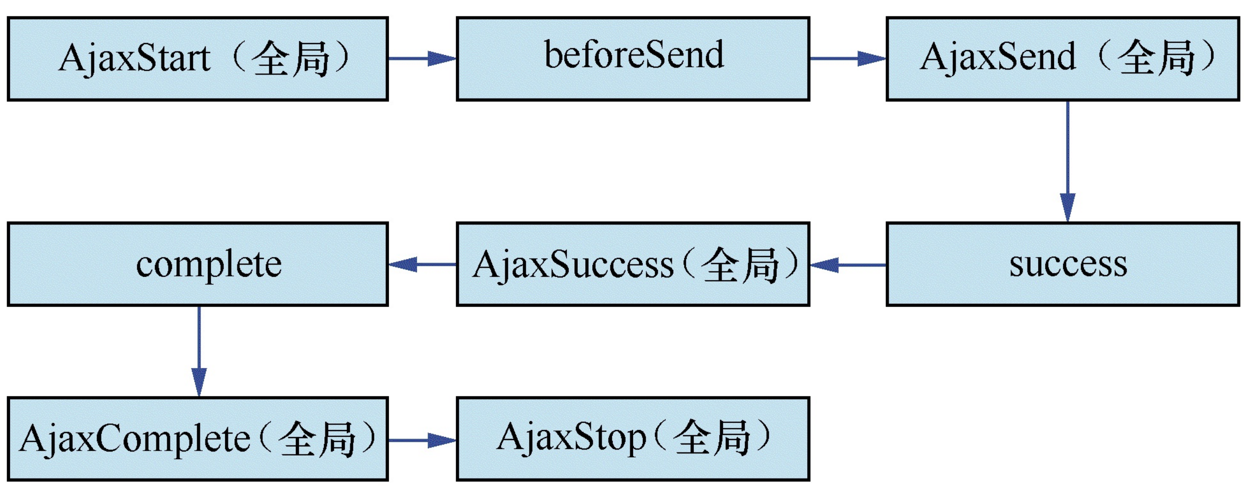


图13-4

2．失败的Ajax请求的事件流（见图13-5）

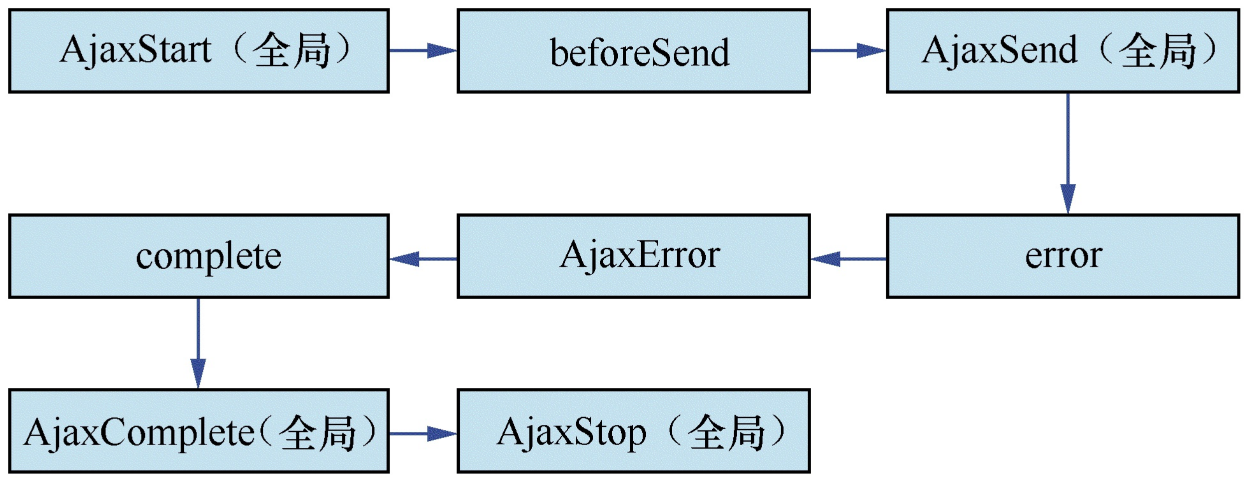


图13-5

https://github.com/ForbesLindesay/ajax基于早期的jQuery.ajax设计出来，用法与jQuery一致，可惜不支持Promise。PC上要单独使用Ajax时可以使用此库。

https://github.com/visionmedia/superagent由大牛亲自打造，browser/nodejs通用，插件丰富，可惜不支持Promise及兼容性差（IE0＋）。

https://github.com/mzabriskie/axios一个成名更早的Ajax库，支持Promise，架构优美，兼容性好

（IE8＋）。在业务线使用时，这个库也是推荐使用。

https://github.com/ded/reqwest与jQuery.ajax基本兼容，支持Promise，兼容性优秀（IE6＋），设计精巧，代码量少（依赖于xhr2库）。

https://github.com/pyrsmk/qwest从名字可以看出，它是reqwest的模仿者，优点应该与它与一致，但不能运行于Node.js。它行数基于更少。它需要依赖一个不怎么标准的Promise。

13.8　fetch，下一代Ajax

Ajax自2005年出现在公众面前，已经有十多年，算是一个很长寿的技术方案了。而它的取替点，则安照W3C一惯的手段，从社区成熟的方案标准化。fetch，大抵可以认为是jQuery.ajax的官方版。

脱胎于微软私有的异步请求方案XMLHttpRequest，其运作方式于现在的眼光来看，已经有点落后了。它是使用on×××来添加回调，是典型的事件模式。而异步事件机制的并发竞争情况，它很难处理，一不心小就出回调地狱，于是有了jQuery的Deferred及后来的Promise。

我们比较一下这两者的写法吧。

```
//ajax
var xhr = new XMLHttpRequest();
xhr.open('GET', url);
xhr.responseType = 'json';
```

```
xhr.onload = function() {
  console.log(xhr.response);
};
xhr.onerror = function() {
  console.log("Oops, error");
};
xhr.send();
//fetch
fetch(url).then(function(response) {
  return response.json();
}).then(function(data) {
  console.log(data);
}).catch(function(e) {
  console.log("Oops, error");
});
```

从形式上看，fetch是比Ajax简单些，但实际不然。fetch需要4个东西配合一起使用。我们看一下API。

首先要引入fetch库，这东西的兼容性还是非常差，如图13-6所示。

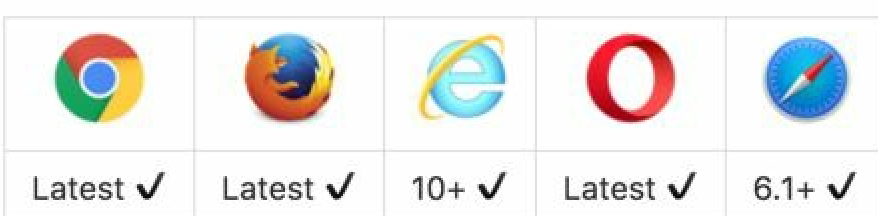


图13-6

它总共有4个对象，**fetch**、**Request**、**Headers**、Response。

`fetch` 是入口函数，它要求传入一个`url` 或一个**Request** 实例，后面再跟一个可选的配置对象。配置对象与Request对象可以指定一个`headers` 属性，这个对象是一个对象，可以是普通JavaScript对象，也可以是一个Headers实例，最后它会在第一个then方法中返回**Responses** 实例。这样，4个东西就串起来了。

下面是官方的几个简单例子。

1. 返回文本

```
fetch('/users.json').then(function(response) {
  console.log(response.headers.get('Content-Type'))
  console.log(response.headers.get('Date'))
  console.log(response.status)
  console.log(response.statusText)
})
```

2. 返回JSON

这个见上面的例子。

3. 抽取状态码等元信息

```
fetch('/users.html')
  .then(function(response) {
  //无论后端返回什么，response的text方法会将它转换为文本
    return response.text()
  }).then(function(body) {
    document.body.innerHTML = body
  })
```

4. 发送POST请求

```
var form = document.querySelector('form')

fetch('/users', {
  method: 'POST',
  body: new FormData(form)
})
```

5. 上传文件

```
var input = document.querySelector('input[type="file"]')

var data = new FormData()
data.append('file', input.files[0])
data.append('user', 'hubot')

fetch('/avatars', {
  method: 'POST',
  body: data
}).then(function(){/***/})
```

或者使用Request实例上传文件。

```
var myImage = document.querySelector('img');
var myRequest = new Request('flowers.jpg');
fetch(myRequest).then(function(response) {
  return response.blob();
}).then(function(response) {//图片预览
  var objectURL = URL.createObjectURL(response);
  myImage.src = objectURL;
});
```

使用header实例设置请求头。

```
var myImage = document.querySelector('img');
var myHeaders = new Headers();
myHeaders.append('Content-Type', 'image/jpeg');

var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };
var myRequest = new Request('flowers.jpg');

fetch(myRequest,myInit).then(function(response) {
  ...
});
```

Request对象拥有如下配置项。

- method：请求类型，GET, POST, PUT, DELETE, HEAD。
- url：请求地址，URL of the request。
- headers：关联的Header对象。
- referrer：referrer。
- mode：请求的模式，主要用于跨域设置，cors, no-cors, same-origin。
- credentials：是否发送Cookie omit, same-origin。
- redirect：收到重定向请求之后的操作，follow, error, manual。
- integrity：完整性校验。
- cache：缓存模式（default, reload, no-cache）。

Response对象拥有如下方法（我们不用关心它是如何构建的）

The Response also provides the following methods。

- clone()：复制自身。
- error()：获取错误详情的Promise。
- redirect()：跳转。
- arrayBuffer()：返回一个Arraybuffer数据的Promise。
- blob()：返回一个Blob数据的Promise。
- formData()：返回一个FormData数据的Promise。
- json()：返回一个JSON数据的Promise。
- text()：返回一个字符串数据的Promise。
- status：状态码（ex: 200, 404, etc．）。
- ok：是否成功响应（status in the range 200-299）。
- statusText：status code (ex: OK) 。
- headers：响应头。

用法是简单的，但是如果想在低版本浏览器下使用（主要是IE6～IE8），还是很艰难。总不能用一项新技术，就不管旧客户嘛，因此我们像叠积木

一样堆砌polyfill，基本可以支持IE8+。

- 引入ES5的polyfill: es5-shim, es5-sham。
- 引入Promise的polyfill: es6-promise。
- 引入fetch探测库：fetch-detector。
- 引入fetch的polyfill: fetch-ie8。
- 可选：如果你还使用了jsonp，引入fetch-jsonp。

6．fetch常见“坑”

fetch请求默认是不带cookie的，需要设置fetch(url, {credentials: 'include'})。服务器返回400，500错误码时并不会reject，只有网络错误导致请求不能完成时，fetch才会被reject。

7．IE使用策略

IE8、IE9的XHR不支持CORS跨域，虽然提供XDomainRequest，但这个东西就是玩具，不支持传Cookie！如果接口需要权限验证，还是乖乖地使用

jsonp吧，推荐使用fetch-jsonp。

在本书成书时，github上有关fetch的polyfill还是很少，但当作为一个潮流，fetch取替Ajax只是时间的问题。如果大家还坚持自己写Ajax库，建议参考fetch的API来构建。

第14章　动画引擎



浏览网页时，我们经常被一些创意所感动。无论是极具视觉冲击的动画，还是很平缓但舒适的细微变动，都是这东西所创造的。动画引擎听起来是很高级的东西，但原理却很简单。

以前听说日本的动画是这样做出来：一个人在

一本空白的书中画了许多画，这些画每一张与上一张都有细微的差异，然后快速翻动书页，就看起来像动一样。动画只不过是我们眼睛的残影，叫做视觉暂留现象。这里有两个关键字：差异与快速。在网页中，控制样式的任务已经交由CSS掌控，让JavaScript第一次拥有时间处理的API，setTimeout与setInterval早在CSS诞生前已出现。CSS可划分两种，一种的值近似一个无限集合，一种只有寥寥几个值。能够度量与变化的也是那个无限集，就是像颜色值，red、yellow、black，我们也可以转换为RGB进行计算。我们可以在定时器里面，每隔20ms～30ms改变这些样式值，于是就有了动画。改变宽高，就叫缩放；改变坐标，就叫位移；改变坐标轴，就叫旋转；改变透明度，就叫淡入淡出……

14.1　动画的原理

在标准浏览器中，可计算的样式值基本上它已经为你转换好，如width、height、margin-x、border-x-width、padding-x，这些样式的单位为px。color、background-color则被分解为RGB，这个很容易就格式化为一个数组，透明度就自不用说。不过，最恶心的是新引进的变形样式transfrom，它有两类传值方式，一种面向人类，是rotate()、skew()、scale()、translate()，分别还有x、y之分，比如rotateX()和rotateY()，以此类推；一类面向计算机，传入矩阵进去，matrix()，无论是传什么，它最后都转成矩阵。

如果换是旧版本IE，那么就得自己转了，比如你原来的单位是em, currentStyle会返回em，原来填的颜色是red，它不会返回rgb(255,0,0)给你。

因此搞动画引擎的第一步是设法获得元素的精确样式值，这个笔者在样式模块这一章有介绍，那里给出的$.css方法基本上除颜色值都是已转换好的。

现在我们尝试让一个方块动起来吧。动起来，换言之就是改变位置，在CSS就是对应top与left，当然你还可以用margin-left，现在我们只讲最通用的方式。要想用top与left还需要让它相对定位或绝对定位。通常的做法是父元素相对定位，成为包含块，子元素绝对定位。只有定位了，top与left才不会返回auto，而是返回可计算的像素值。首先要得到它原来的位置。简单起见，先实现平移，那么只需要取left，读者可以在Firefox下做实现，直接使用getComputedStyle，然后让用户传结束位置，起始位置与结束位置之间的距离，就是要一点点操作的变量。动画还涉及时间，一个时长，也就是动画执行的总时间，另一个是每次变动的相隔时间，这个

通常由引擎决定，当然也可以暴露出来，它有个学名叫做fps。

fps通俗地说，叫刷新率，在1秒内更新多少次画面。根据人眼睛的视觉停留效应，若前一幅画像留在大脑中的印象还没消失，后一幅画像就接踵而至，而且两幅画面间的差别很小，就会有“动”的感觉。那么停留多少毫秒最合适呢？我们不但要照顾人的眼睛，还要顾及一下显示器的显示速度与浏览器的渲染速度。根据外国的统计，25毫秒为最佳数值。其实，这个数值我们应该当作常识来记住。联想一下，日本动画好像有个规定是1秒30张画，中国的是1秒24张。用1秒去除以张数，就得到每幅画面停留的时间。日本的那个27.77毫秒已经很接近我们的25毫秒了，因为浏览器的渲染速度明显不如电视机的渲染速度，尤其IE6这个拉后腿的。距离与时间出来了，那么我们再求一下速度就行了。

为此我们建一个新页面，里面有一个方块，它位于包含块这个轨道上。它要从这一端跑到另一个端。动画时间为2秒，fps为30帧。

```
<style type="text/css">
    #taxiway{
        width:800px;
        height:100px;
        background:#E8E8FF;
        position: relative;
    }
    #move{
        position: absolute;
        left:0px;
        width:100px;
        height:100px;
        background:#a9ea00;
    }
</style>
<div id="taxiway">
      <div id="move" ></div>
</div>
<script>
    window.onload = function() {
        var el = document.getElementById("move");
        var parent = document.getElementById("taxiway")
        var distance = parent.offsetWidth - el.offsetWidth;
            //总移动距离
        var begin = parseFloat(window.getComputedStyle(el, null
).left);    //开始位置
        var end = begin + distance;                        //结束位置
        var fps = 30;                                      //刷新率
        var interval = 1000 / fps;                         //每相隔多
少ms刷新一次
        var duration = 2000;//时长
        var times = duration / 1000 * fps;                 //一共刷新
这么多次
        var step = distance / times;//每次移动多少距离
        el.onclick = function() {
            var now = new Date
```

```
            var id = setInterval(function() {
                if (begin >= end) {
                    el.style.left = end + "px";
                    clearInterval(id);
                    alert(new Date - now)
                } else {
                    begin += step;
                    el.style.left = begin + "px"
                }
            }, interval)
        }
    }
</script>
```

上例中，我们使用了最简单的累加来实现。现在我们改写一下，加入进度这个变量，让其更具广泛性。

```
el.onclick = function() {
    var beginTime = new Date
    var id = setInterval(function() {
        var t = new Date - beginTime;//当时已用掉的时间
        if (t >= duration) {
            el.style.left = end + "px";
            clearInterval(id);
            alert(t)
        } else {
            var per = t / duration;//当前进度
            el.style.left = begin + per * distance + "px"
        }
    }, interval)
}
```

如果我们能随意控制per这个数值，那么就能轻易实现加速或减速。于是乎，人们发明了缓动公式。所谓缓动公式其实来自数学上的三角函数、二次项方程式、高阶方程式，它们最初由Flash界的Robert Penner整理收来。有了缓动公式，我们就能轻松模拟现实中加速、减速、急刹车、重力、摇摆、弹簧、来回弹动等效果。

14.2　缓动公式

经过这么多年的发展，缓动公式的各项规范都稳定下来。虽然现在还有人源源不断地发掘出新式，但一般业务中，绝对多数人都是用默认的easeIn或linear。因此如果对库的大小有顾虑，那么就将它们独立成一个模块吧。

现在所有缓动公式，基本上除了linear外（但它也常被称为easeNone），它们都以ease开头命名。

添加3种后缀，In表示加速，Out表示减速，InOut表示加速到中途又开始减速，于是就有easeIn、easeOut、easeInOut之分。如果单是这样命名，说明，它们没有介入高阶函数与三角函数，像linear就是匀速。

然后再以实现方式与指数或开根进行区分。Sine表示由三角函数实现，Quad是二次方，Cubic是三次方，Quart是四次方，Quint是五次方，Circ使用开平方根的Math.sqit，Expo使用开立方根的Math.pow, Elastic则是结合三角函数与开立三方根的初级弹簧效果，Back是使用了一个1.70158常数来计算的回退效果，Bounce则是高级弹簧效果，如图14-1所示。

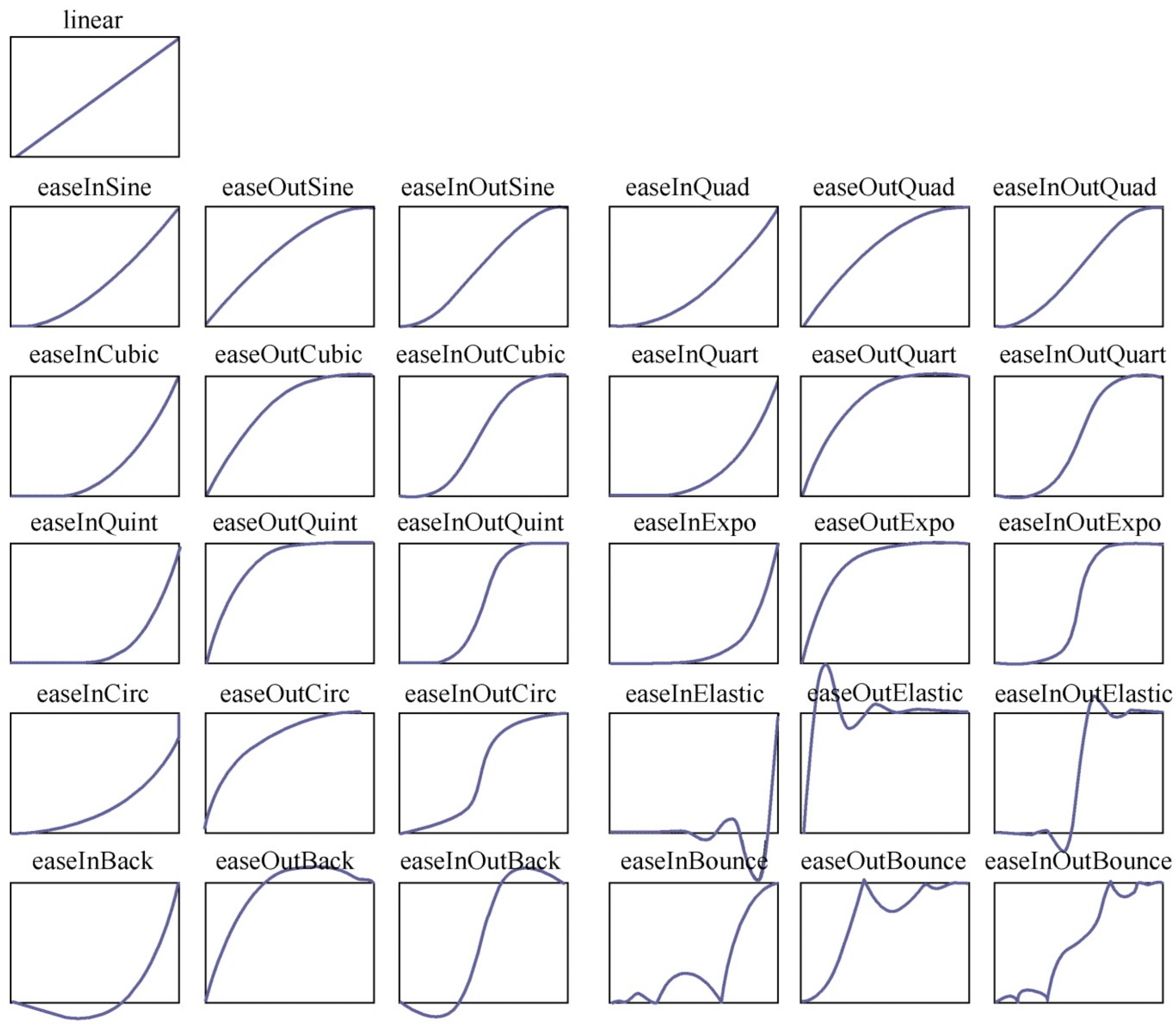


图14-1

图片出自

http://hosted.zeh.com.br/tweener/docs/en-us/misc/transitions.html

这些公式我们可以在AS库，或

jquery.easing.js、mootools库里面找到，基本上大同小异。其中jquery.easing.js最规范，mootools使用循环生成方式实现，代码最简。读者想实现自己的动画库，可以参考这两者。

jQuery标准库里面只有两条：

```
linear: function( p ) {
    return p;
},
swing: function( p ) {
    return 0.5 - Math.cos( p*Math.PI ) / 2;
}
```

但显然与其他缓动库的参数形式不一样：

```
var easing = {
    easeInQuad: function( t, b, c, d) {
        return c * (t /= d) * t + b;
    },
    easeOutQuad: function( t, b, c, d) {
        return -c * (t /= d) * (t - 2) + b;
    },
    easeInOutQuad: function( t, b, c, d) {
        if ((t /= d / 2) < 1)
            return c / 2 * t * t + b;
        return -c / 2 * ((--t) * (t - 2) - 1) + b;
    }
    //……略
}
```

这是因为jQuery在上面的计算过程已经做了这一步。我们看这4个参数分别代表什么。

- T：timestamp，指缓动效果开始执行到当前帧所经过的时间段，单位ms。
- B：beginning val，起始值。
- C：change，要变化的总量。
- D：duration，动画持续的时间。

返回的是直接可用的数值，或许我们只要加个单位进行赋值就行了。

而jQuery那一个参数的风格，其实是当前时间减去动画开始时间除以总时间的比值，一个0到1的小数，它用于乘以总变化量，然后加上起始值，就是现在此样式的情况。下面是缓动的应用，还是使用上例。

```
window.onload = function() {
```

```
    var el = document.getElementById("move");
    var parent = document.getElementById("taxiway")
    var change = parent.offsetWidth - el.offsetWidth; //总变化量
    var begin = parseFloat(window.getComputedStyle(el, null).le
ft);//起始值
    var end = begin + change;                                  //结束值
    var fps = 30;                                              //刷新率
     var interval = 1000 / fps;                                //每相隔多
少ms刷新一次
    var duration = 2000;                                       //时长
    var bounce = function(per) {                               //缓动公式
，弹簧
        if (per < (1 / 2.75)) {
            return (7.5625 * per * per);
        } else if (per < (2 / 2.75)) {
            return (7.5625 * (per -= (1.5 / 2.75)) * per + .75)
;
        } else if (per < (2.5 / 2.75)) {
            return (7.5625 * (per -= (2.25 / 2.75)) * per + .93
75);
        } else {
            return (7.5625 * (per -= (2.625 / 2.75)) * per + .9
84375);
        }
    }
    el.onclick = function() {
        var beginTime = new Date
        var id = setInterval(function() {
            var per = (new Date - beginTime) / duration;//进度
            if (per >= 1) {
                el.style.left = end + "px";
                clearInterval(id);
            } else {
                el.style.left = begin + bounce(per) * change +
"px";
            }
        }, interval)
    }
}
```

点击它就发现方块到终点后还弹回来，再过

去，又弹回来，渐渐平息……

14.3 jQuery.animate

现在我们看如何写动画引擎。由于选择器的流行，注定这个API到用户这里也是集化操作，一下子处理N个元素。但这也无所谓，关键是函数名与如何传参。

jQuery的API易用性很好，我们看一下它的animate，有2种用法。

```
.animate( properties [, duration ] [, easing ] [, complete ] )
.animate( properties, options )
```

第一个参数恒为要进行动画的属性的映射，在第一种情况下，其他参数都是可选的，因为duration除了slow、fast、default这3个字符串就是数字，easing为特殊的缓动公式的名字，complete是函数，options是对象。不过尽管说得轻松，jQuery在这里也花了几十行进行转换。最后转换两个对象的

形式其实与后来出现的CSS3 keyframe animation的定义非常吻合。

我们还是利用上例的方块，加个类名animate，就能看到效果了。

```
.animate {
    animation-duration: 3s;
    animation-name: slidein;
    animation-timing-function:ease-in-out;
    animation-fill-mode: forwards
}

keyframes slidein {
    from {
        left: 0%;
        background: white;
   }
   to {
        left: 700px;
        background: red;
   }
}
```

这个动画分为两部分：第一个部分是普通的样式规则，用于描述动画所需时长，缓动公式，结束后保留状态，重复多少次，及关键帧动画的引用名字（slidein）；第二个部分是关键帧规则。这里只

插入两个关键帧，实际上可以插入更多，最开始与最尾的可以用from、to命名。其实当你用JavaScript去取时，它们都会转换为百分比，0%或100%。其中以最尾的最重要，没有浏览器会补上，它就是对应jQuery.animate方法的第一个参数。至于初始值，浏览器会自动计算，如果我们是使用纯JavaScript实现方式，它们就要我们来计算了。动画引擎的强大与否，这时就取决于CSS模块是否强大了。animate方法的第二个参数等对应animation-duration、animation-timing-function、animation-fill-mode等设置。

除此之外，jQuery还提供了一个queue参数，目的让作用于同一个元素的动画进行排队，先处理完这个再处理后一个。要不，同时运行于浏览器中的定时器就太多了。加之集化操作，会把它放大化，队列的重要性就更突出。为此jQuery把这个参数默认为true。

从实现上看，jQuery的队列是放在元素对应的缓存体上，里面是一个Promise对象，complete之后，会自动弹出下一个动画对象。所有动画对象都有自己的setInterval驱动。在YUI、kissy、mass Framework则有一个中央队列，所有不排队的动画全部放在这数组中，然后有一个setInterval来驱动它们，排队的动画作为它的兄弟的属性而存在（有的中央队列可能是一条二维数组），当前面的动画执行完，排队的动画就会翻身。

14.4 mass Framework基于JavaScript的动画引擎

现在我们看一下，动画引擎是如何做的。首先，我们搞一个中央队列（也叫时间轴，我们在里面插入关键帧，两个关键帧之间就是我们的补间动画），其实就是一个数组。只要它里面有一个元素，它就驱动setInterval执行动画。如果动画执行完毕，它就会删掉其node属性，并且从数组中删除此元素。检测一下数组还有没有元素，没有就clearInterval，否则就继续。

```
https://github.com/RubyLouvre/mass-Framework/blob/master/fx.js
var timeline = $.timeline = [];                     //时间轴
function insertFrame(frame) {                       //插入包含关键帧原始信
息的帧对象
    if (frame.queue) {                              //如果指定要排队
        var gotoQueue = 1;
        for (var i = timeline.length, el; el = timeline[--i];)
{
            if (el.node === frame.node) {           //第一步
                el.positive.push(frame);            //子队列
                gotoQueue = 0;
                break;
            }
        }
        if (gotoQueue) {                            //第二步
```

```
            timeline.unshift(frame);
        }
    } else {
        timeline.push(frame);
    }
    if (insertFrame.id === null) {//只要数组中有一个元素就开始运行
        insertFrame.id = setInterval(deleteFrame, 1000 / $.fps)
;
    }
}
insertFrame.id = null;
```

具体如图14-2所示。

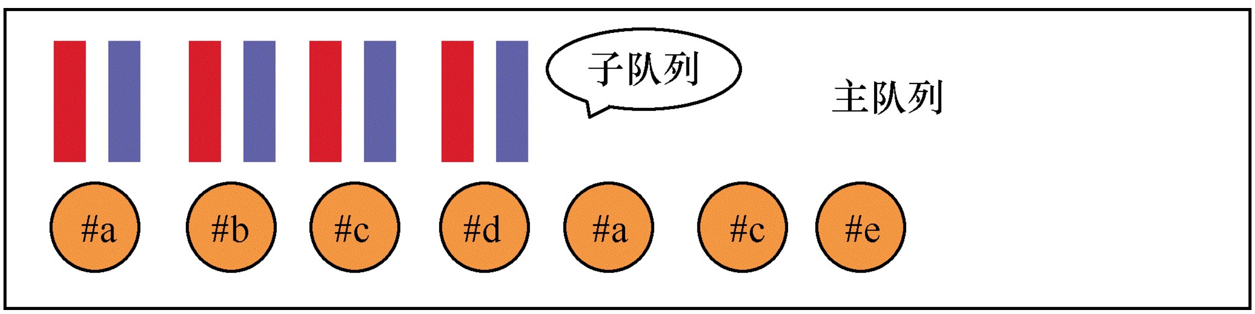


图14-2

主队列的动画是立即执行的，一个元素可以对应多个动画，比如它的宽与高与背景色都要同时改变。在上图中，#a元素就有两个同时进行的动画。但在某些场合，我们要求一个方块像星星一样一闪一闪地眨眼睛，就需要子队列了。子队列是放置等

待执行的动画，只有前面的动画执行完毕，才执行它们。在CSS3中，有animation-fill-mode，能够轻松实现倒带 的效果。

```
<div id="test" class="animate" ></div>
<style type="text/css">
    #test{
        width:100px;
        height:100px;
        background:blue;
    }
  /* 此动画先放大再缩回原状*/
    .animate{
        animation-duration:3s;
        animation-name: cycle;
        animation-iteration-count:2;
        animation-direction:alternate;
    }
    @keyframes cycle{
        to{
        width:200px;
        height:200px;
    }
    }
</style>
```

用JavaScript实现就是，把第一帧与最后一帧调换。我们把这些动画放到negative子队列中。如果不是倒带就放在positive队列。

```
var effect = $.fn.animate = $.fn.fx = function(props) {
```

```
    //将多个参数整成两个，第一参数暂时别动
    var opts = addOptions.apply(null, arguments), p
    //第一个参数为元素的样式，我们需要将它们从CSS的连字符风格统统转为驼峰
风格
    //如果需要私有前缀，也在这里加上
    for (var name in props) {
        p = $.cssName(name) || name;
        if (name !== p) {
            props[p] = props[name];    //添加borderTopWidth、sty
leFloat
            delete props[name];        //删掉border-top-width、f
loat
        }
    }
    for (var i = 0, node; node = this[i++];) {
        //包含关键帧的原始信息的对象到主队列或子队列
        insertFrame($.mix({
            positive: [],              //正向队列
            negative: [],              //外队队列
            node: node,                //元素节点
            props: props               //@keyframes中要处理的样式
集合
        }, opts));
    }
    return this;
}
```

接着我们看deleteFrame方法，它的任务就是把已经完成或强制完成的动画从主队列中删掉。

```
function deleteFrame() {
        var i = timeline.length;
        while (--i >= 0) {
            if (!timeline[i].paused) { //如果没有被暂停
                if (!(timeline[i].node && enterFrame(timeline[i
], i))) {
                     timeline.splice(i, 1);
                }
```

```
            }
        }
        timeline.length || (clearInterval(insertFrame.id), inse
rtFrame.id = null);
}
```

那么如果我们要添加关键帧呢？当然是使用animate方法（它也有个更短的别名叫fx），参数与jQuery一样多态化，那也意味着我们需要花些精力调整它们成可用状态。

调整用户传参需要用到如下这么多代码。

```
function addOptions(properties) {
    if (isFinite(properties)) {//如果第一个为数字
        return {
            duration: properties
        };
    }
    var opts = {};
    //如果第二参数是对象
    for (var i = 1; i < arguments.length; i++) {
        addOption(opts, arguments[i]);
    }
    opts.duration = typeof opts.duration === "number" ? opts.du
ration : 400;
    opts.queue = !!(opts.queue == null || opts.queue); //默认进行
排队
    opts.easing = $.easing[opts.easing] ? opts.easing : "swing"
;
    return opts;
}
function addOption(opts, p) {
```

```
    switch ($.type(p)) {
        case "Object":
            addCallback(opts, p, "after");
            addCallback(opts, p, "before");
            $.mix(opts, p);
            break;
        case "Number":
            opts.duration = p;
            break;
        case "String":
            opts.easing = p;
            break;
        case "Function":
            opts.complete = p;
            break;
    }
}
function addCallback(target, source, name) {
    if (typeof source[name] === "function") {
        var fn = target[name];
        if (fn) {
            target[name] = function(node, fx) {
                fn(node, fx);
                source[name](node, fx);
            };
        } else {
            target[name] = source[name];
        }
    }
    delete source[name];
}
```

然后才进入到我们刚才的insertFrame方法，insertFrame会间接调用enterFrame方法。由于它是在setInterval内运行的，因此它才是动画真正的实现者。

```
function enterFrame(fx, index) {
    //驱动主队列的动画实例进行补间动画(update)
    //并在动画结束后，从子队列选取下一个动画实例取替自身
    var node = fx.node,
            now = +new Date;
    if (!fx.startTime) {                                        //第一帧
        callback(fx, node, "before");                           //动画开始
前做些预操作
        fx.props && parseFrames(fx.node, fx, index);   //分解原始
材料为关键帧
        fx.props = fx.props || [];
        AnimationPreproccess[fx.method || "noop"](node, fx); /
/parse后也要做些预处理
        fx.startTime = now;
    } else {  //中间自动生成的补间
        var per = (now - fx.startTime) / fx.duration;
        var end = fx.gotoEnd || per >= 1;//gotoEnd可以被外面的sto
p方法操控,强制中止
        var hooks = effect.updateHooks;
        if(fx.update){
            for (var i = 0, obj; obj = fx.props[i++]; ) {// 处理
渐变
                (hooks[obj.type] || hooks._default)(node, per, e
nd, obj);
            }
        }
        if (end) { //最后一帧
            callback(fx, node, "after");                        //动画结束
后执行的一些收尾工作
            callback(fx, node, "complete");                     //执行用户
回调
            if (fx.revert && fx.negative.length) {      //如果设置
了倒带
                Array.prototype.unshift.apply(fx.positive, fx.n
egative.reverse());
                fx.negative = [];                               // 清空负
向队列
            }
            var neo = fx.positive.shift();
            if (!neo) {
                return false;
            }//如果存在排队的动画,让它继续
            timeline[index] = neo;
            neo.positive = fx.positive;
            neo.negative = fx.negative;
```

```
        } else {
            callback(fx, node, "step");                //每执行一帧调用的回调
        }
    }
    return true;
}
```

这里面涉及几个辅助函数，callback用于执行回调，包括内部用的before与after，用户设置的step与complete。

```
function callback(fx, node, name) {
    if (fx[name]) {
        fx[name](node, fx);
    }
}
```

parseFrames是相当复杂庞大的，这里笔者就不贴出来了。总而言之，就是从已有的材料中分成两个关键帧，每个关键帧包括样式名、缓动公式（之前只是名字）、开始值、结束值、单位与类型。类型分为3种，颜色值、滚动与默认处理，根据它们，程序会选用不同的钩子函数进行分解、刷新。

```
effect.updateHooks = {
    _default: function(node, per, end, obj) {
        $.css(node, obj.name, (end ? obj.to : obj.from + obj.ea
sing(per) *
          (obj.to - obj.from)) + obj.unit)
    },
    color: function(node, per, end, obj) {
        var pos = obj.easing(per),
                rgb = end ? obj.to : obj.from.map(function(from
, i) {
            return Math.max(Math.min(parseInt(from + (obj.to[i]
 - from) * pos, 10), 255), 0);
        });
        node.style[obj.name] = "rgb(" + rgb + ")";
    },
    scroll: function(node, per, end, obj) {
        node[obj.name] = (end ? obj.to : obj.from + obj.easing(
per) * (obj.to - obj.from));
    }
};
```

这里可留意一下颜色值的刷新函数，它是转换为RGB的形式，再变成数组。刷新函数都有个end参数，用于立即跳到最后一帧。

下面是分解颜色值的代码，只处理16进制、RGB与颜色名这3种。难点在于IE的颜色转化。

```
var colorMap = {
    "black": [0, 0, 0],
    "gray": [128, 128, 128],
    "white": [255, 255, 255],
    "orange": [255, 165, 0],
```

```
    "red": [255, 0, 0],
    "green": [0, 128, 0],
    "yellow": [255, 255, 0],
    "blue": [0, 0, 255]
};

function parseColor(color) {
    var value;//在iframe下进行操作
    $.applyShadowDOM(function(wid, doc, body) {
        var range = body.createTextRange();
        body.style.color = color;
        value = range.queryCommandValue("ForeColor");
    });
    return [value & 0xff, (value & 0xff00) >> 8, (value & 0xff0
000) >> 16];
}

function color2array(val) { //将字符串变成数组
    var color = val.toLowerCase(),
            ret = [];
    if (colorMap[color]) {
        return colorMap[color];
    }
    if (color.indexOf("rgb") === 0) {
        var match = color.match(/(\d+%?)/g),
                factor = match[0].indexOf("%") !== -1 ? 2.55 :
1;
        return (colorMap[color] = [parseInt(match[0]) * factor,
 parseInt(match[1]) * factor, parseInt(match[2]) * factor]);
    } else if (color.charAt(0) === '#') {
        if (color.length === 4)
            color = color.replace(/([^#])/g, '$1$1');
        color.replace(/\w{2}/g, function(a) {
            ret.push(parseInt(a, 16));
        });
        return (colorMap[color] = ret);
    }
    if (window.VBArray) {
        return (colorMap[color] = parseColor(color));
    }
    return colorMap.white;
}
$.parseColor = color2array;
```

在enterFrame方法中有一个预处理的过程，主要是用于show、hide等方法。AnimationPreproccess里面有4方法：noop、show、hide、toggle。

noop 是不处理。

show 是将隐藏的元素的display改为block等值（有时未必改为block，像li、td、tr、tbody、table都有默认的display值，如果强行改block，布局就会走形；对于内联元素，span、em等，要使用它们进行缩放操作，则要设置为inline-block，但众所周知，旧版本IE要开启hasLayout才能生效，这又是一番周折）。

hide 是将显示的元素隐藏起来，由于它对应的动画效果是从大到小，这时进行动画的那个元素的子元素可能会超出父元素的大小，被挤出来。因此我们需要强制设置它的overflow为hidden，在动画结束后才还原（因为这时display为none，你怎么玩

也没人知道）。此外，要还原的样式值还有宽高、边框、补白、外界、透明度等值，这方便我们在show→hide→show这样的连续动画中能运行起来。

toggle 就是对隐藏元素进行show操作，显示元素进行hide操作。

```
var AnimationPreproccess = {
    noop: $.noop,
    show: function(node, frame) {
        if (node.nodeType === 1 && $.isHidden(node)) {
            var display = $._data(node, "olddisplay");
            if (!display || display === "none") {
                display = $.parseDisplay(node.nodeName);
                $._data(node, "olddisplay", display);
            }
            node.style.display = display;
            if ("width" in frame.props || "height" in frame.pro
ps) {
                //如果是缩放操作
                //修正内联元素的display为inline-block，让其可以进行w
idth/height的动画渐变
                if (display === "inline" && $.css(node, "float"
) === "none") {
                    if (!$.support.inlineBlockNeedsLayout) { //
W3C
                        node.style.display = "inline-block";
                    } else { //IE
                        if (display === "inline") {
                            node.style.display = "inline-block"
;
                        } else {
                            node.style.display = "inline";
                            node.style.zoom = 1;
                        }
                    }
```

```
                }
            }
        }
    },
    hide: function(node, frame) {
        if (node.nodeType === 1 && !$.isHidden(node)) {
            var display = $.css(node, "display"),
                    s = node.style;
            if (display !== "none" && !$._data(node, "olddispla
y")) {
                $._data(node, "olddisplay", display);
            }
            var overflows;
            if ("width" in frame.props || "height" in frame.pro
ps) {
                //如果是缩放操作
                //确保内容不会溢出,记录原来的overflow属性
                //因为IE在改变overflowX与overflowY时，overflow不会
发生改变
                overflows = [s.overflow, s.overflowX, s.overflo
wY];
                s.overflow = "hidden";
            }
            var fn = frame.after || $.noop;
            frame.after = function(node, fx) {
                if (fx.method === "hide") {
                    node.style.display = "none";
                    for (var i in fx.orig) { //还原为初始状态
                        $.css(node, i, fx.orig[i]);
                    }
                }
                if (overflows) {
                    ["", "X", "Y"].forEach(function(postfix, in
dex) {
                        s["overflow" + postfix] = overflows[ind
ex];
                    });
                }
                fn(node, fx);
            };
        }
    },
    toggle: function(node, fx) {
        $[$.isHidden(node) ? "show" : "hide"](node, fx);
    }
```

```
};
```

至此，整个动画引擎就完成了。而那些show、hide、toggle、slideUp、slideDown、slideToggle、show、hide、toggle、slideUp、slideDown、slideToggle、fadeIn、fadeToggle、fadeOut等特效，其实就是利用animate这个主函数，预先传些样式进去。比如fadeIn就是将透明度逐渐由0变成1，fadeOut就是从1到0，slideUp就是将高度逐渐减为0，slideDown就是从0逐渐还原为之前的高度，show与hide则是同时对宽高边界边框补白等盒子属性与透明度进行动画。

Prototype.js的重要特性script.aculo.us可能是JavaScript最强大的动画框架了，它的许多特效都是其他动画库的重要参考，如表14-1所示。

表**14-1**

script.aculo.us	**jQuery或 jQuery UI**	说　　明
Appear	fadeIn	淡入
Fade	fadeOut	淡出
Puff	puff	整体扩大一倍并同时进行淡出
DropOut	drop	整体往下堕落并同时进行淡出，最后不占空间
Shake	shake	像蛇信子一样左右震荡几下
SwitchOff	clip	像电视关闭时的收场动画一样两边往中间叠起一线消失
BlindDown	slideDown	像展开卷轴般从上到下呈现元素
BlindUp	slideUp	像收起卷轴般从下到上折起元素
SlideDown	blind	像升降机从我们眼前落下，jQuery UI的blind、slide恰好与script.aculo.us的相反
SlideUp	Blind	像升降机从我们眼前升起
Pulsate	pulsate	闪动几下
Squish	show	四边往左上角收缩，有时类似jQuery的show效果，但jQuery同时进行淡出
Fold	fold	先是高度从下往上收起，然后宽度从右到左收起

[illegible]	[illegible]	[illegible]
Grow		从中间一点向四边扩张
Shrink	scale	四边往中间一点收缩

基本上script.aculo.us能做的，jQuery UI也能轻松实现。不过jQuery UI每个子特效设计得非常强大，因此体积也不遑多让。

14.5　requestAnimationFrame

如果一个页面运行许多定时器，那么你无论怎么优化，最后肯定是超过指定时间才能完成动画，定时器越多，延时越严重。为此，YUI、kissy、mass等采用中央队列的方式，将定时器减少至一个。浏览器厂商也不是吃素的，最早Firefox也想到这点，早在4.0时就推mozRequestAnimationFrame。当然，它与现在的标准相差很远。它不是个定时器，甚至也不能传递回调，它只是用于触发MozBeforePaint这个私有事件。由于是浏览器在维护队列，它内部掌握DOM渲染，事件队列排队等情况，因此它大抵能保证fps在60帧上下。

```
<script>
  /*firefox4-10*/
  var startTime,
          duration = 3000;
  function animate(event) {
      var now = event.timeStamp;
      var per = (now - startTime) / duration;
      if (per >= 1) {
          window.removeEventListener('MozBeforePaint', animate,
```

```
 false);
      } else {
          document.getElementById("test").style.left = Math.rou
nd(600 * per) + "px";
          window.mozRequestAnimationFrame();
      }
  }
  function start() {
      startTime = Date.now();
      window.addEventListener('MozBeforePaint', animate, false)
;
      window.mozRequestAnimationFrame();
  }
</script>
<button onclick="start()">点我</button>
<div id="test" style="position: absolute; left: 10px; backgroun
d: blue;"> go! </div>
```

谷歌发现这个思路不错，立即整到自己的Chrome中去，但相对而言，精简很多，与最后定案的标准差距很少。webkitRequestAnimationFrame 有点像定时器，第一个是回调，第二个可选，传执行动画的元素节点进去，返回一个ID，然后允许像clearTimeout一样有个webkitCancelRequest-AnimationFrame函数进行中止动画。不过名字有点长了，后来模仿Firefox一样使用cancelAnimation Frame，不同的只是前面的私有前缀，即改成

webkitCancelAnimationFrame。

```
<script>
  /*chrome10-23*/
  var startTime,
          duration = 3000;
  function animate(now) {
      var per = (now - startTime) / duration;
    if(per >= 1) {
        window.webkitCancelRequestAnimationFrame(requestID);
    } else {
        document.getElementById("test").style.left = Math.round
(600 * per) + "px";
        window.webkitRequestAnimationFrame(animate);//不断地递归
调用animate
    }
}
function start() {
    startTime = Date.now();
    requestID = window.webkitRequestAnimationFrame(animate);
}
</script>
<button onclick="start()">点我</button>
<div id="test" style="position: absolute; left: 10px; backgroun
d: red;"> go! </div>
```

IE与Opera起步最晚，直到IE10才支持，不过那时标准已经成形，没有私有前缀，也没有兼容性问题。Opera则到12版还不支持。

网上流行一个非常不负责的写法，以为光是换个名字就行。

```
window.requestAnimFrame = (function(){
  return  window.requestAnimationFrame       ||
          window.webkitRequestAnimationFrame ||
          window.mozRequestAnimationFrame    ||
          function( callback ){
            window.setTimeout(callback, 1000 / 60);
          };
})();
```

还有另一个看起来非常周到的写法。

```
(function() {
    var lastTime = 0;
    var vendors = ['webkit', 'moz'];
    for(var x = 0; x < vendors.length && !window.requestAnimati
onFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x]+'Reque
stAnimationFrame'];
        window.cancelAnimationFrame =
          window[vendors[x]+'CancelAnimationFrame'] || window[v
endors[x]+'CancelRequest
          AnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, eleme
nt) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastT
ime));
            var id = window.setTimeout(function() { callback(cu
rrTime + timeToCall); },
              timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
```

```
        };
}());
```

别说早期Firefox不支持传参，它到Firefox11才与webkit那个有点相近，并且才有了mozCancel AnimationFrame。webkit也有bug，它在某个版本中，竟然忘了返回id给我们清除。在另一个版本发现它竟然没有给回调传参。

此外，还有另一个比较流行的版本是这样的，只能执行，不能终止。

```
/* rAF shim. Gist: https://gist.github.com/julianshapiro/949751
3 */
 var rAFShim = (function() {
     var timeLast = 0;

     return window.webkitRequestAnimationFrame || window.mozReq
uestAnimationFrame || function(callback) {
         var timeCurrent = (new Date()).getTime(),
             timeDelta;

         /* Dynamically set delay on a per-tick basis to match
60fps. */
         /* Technique by Erik Moller. MIT license: https://gist
.github.com/paulirish/1579671 */
         timeDelta = Math.max(0, 16 - (timeCurrent - timeLast))
;
         timeLast = timeCurrent + timeDelta;
```

```
        return setTimeout(function() { callback(timeCurrent +
timeDelta); }, timeDelta);
    };
})();
```

下面笔者给出真正可用的兼容版本。

```
// by 司徒正美 基于网友屈屈与月影的版本改进而来
// https://github.com/wedteam/qwrap-components/blob/master/anim
ation/anim.frame.js
function getAnimationFrame() {
    //不存在msRequestAnimationFrame，IE10与Chrome24直接用:requestA
nimationFrame
    if (window.requestAnimationFrame) {
        return {
            request: requestAnimationFrame,
            cancel: cancelAnimationFrame
        }
        //Firefox11-没有实现cancelRequestAnimationFrame
        //并且mozRequestAnimationFrame与标准出入过大
    } else if (window.mozCancelRequestAnimationFrame && window.
mozCancelAnimationFrame) {
        return {
            request: mozRequestAnimationFrame,
            cancel: mozCancelAnimationFrame
        }
    } else if (window.webkitRequestAnimationFrame && webkitRequ
estAnimationFrame(String)) {
        return {//修正某个特异的webKit版本下没有time参数
            request: function(callback) {
                return window.webkitRequestAnimationFrame(
                        function() {
                            return callback(new Date - 0);
                        }
                );
            },
            cancel: window.webkitCancelAnimationFrame ||
              window.webkitCancelRequestAnimationFrame
        }
    } else {
```

```
        var millisec = 25;     //40fps;
        var callbacks = [];
        var id = 0, cursor = 0;
        function playAll() {
            var cloned = callbacks.slice(0);
            cursor += callbacks.length;
            callbacks.length = 0; //清空队列
            for (var i = 0, callback; callback = cloned[i++]; )
 {
                if (callback !== "cancelled") {
                    callback(new Date - 0);
                }
            }
        }
        window.setInterval(playAll, millisec);
        return {
            request: function(handler) {
                callbacks.push(handler);
                return id++;
            },
            cancel: function(id) {
                callbacks[id - cursor] = "cancelled";
            }
        };
    }
}
```

当然，requestAnimationFrame不是没有缺点，它不能控制fps，比如我们做一些慢放动作，许多回调都是做无用功。另一个极端，在动作、枪战、飞车等动态场景时，如果帧数不够高，画面会发虚或模糊。利用原始的setTimeout（在IE9、IE10、Firefox10、Chrome等浏览器中，它的最短时钟间隔

已经压缩至4ms，能轻松跑100帧以上的动画），能让画面更清楚，细节逼真，特写镜头丝丝入扣。

另外，我们还可以尝试一下postMessage这个异步方法，能实现超高度的动画（IE10有setImmediate，速度也相当不错）。下面有个实验，可以看到它们的性能差异。

```
<script>
    var fps_arr, fps_min, fps_max, last_time, loop_iteration;
    var testing = false;
    var fps_label;
    function stop_test() {
        testing = false;
    }
    var init_test = function init_test() {
        fps_arr = [];
        fps_min = 1000;
        fps_max = last_time = loop_iteration = 0;
        if (typeof fps_label === 'undefined') {
            fps_label = document.getElementById('fps_label');
        }
        testing = true;
    }
    function main() {
        var i, fps_avg = 0;
        var now = new Date().getTime();
        if (last_time !== 0 && last_time !== now) {
            var fps = Math.round(1000 / (now - last_time));

            fps_arr.push(fps);
            if (fps_arr.length > 100) {
                fps_arr.shift();
            }
            for (i = 0; i < fps_arr.length; i++) {
```

```
                fps_avg += fps_arr[i];
            }
            fps_avg /= fps_arr.length;
            fps_avg = Math.round(fps_avg);//平均帧数

            if (++loop_iteration > 1) {
                if (fps < fps_min) {
                    fps_min = fps;//最小帧数
                }
                if (fps > fps_max) {
                    fps_max = fps;//最大帧数
                }
            }
            fps_label.innerHTML = fps + ' FPS (' + fps_min + '
- ' + fps_max + ') [平均为 ' +
            fps_avg + ']';
        }
        last_time = now;
    }
    /* Pure Timers */
    function run_timers() {
        main();
        if (testing === true) {
            setTimeout(run_timers, 1);
        }
    }
    window.requestAnimFrame =
            window.requestAnimationFrame ||
            window.webkitRequestAnimationFrame ||
            window.mozRequestAnimationFrame;
    /* requestAnimationFrame */
    function run_raf() {
        main();
        if (testing === true) {
            requestAnimFrame(run_raf, 1);
        }
    }
    /* for loop */
    function run_loop() {
        var iterations = 15;
        while (iterations--) {
            main();
        }
        if (testing === true) {
            setTimeout(run_loop, 1);
```

```
        }
    }
    /* postMessage */
    function run_message() {
        main();
        if (testing === true) {
            window.postMessage('', '*');
        }
    }
    window.addEventListener('message', run_message, false);
</script>
<p id="fps_label"># fps (# - #) [#]</p>
<button onClick="stop_test();">中止</button>

<br /><br />
<strong>Tests</strong><br />
<button onClick="init_test();
        run_timers();">Pure Timers</button>
<button onClick="init_test();
        run_raf();">requestAnimationFrame</button>
<button onClick="init_test();
        run_loop();"> Loop</button>
<button onClick="init_test();
        run_message();">postMessage</button>
```

结果大致如下，视各位的电脑性能而言，如表14-2所示。

表**14-2**

类型	**setTimeout**	**requestAnimationFrame**	**loop**	**postMessage**
平均帧数	200	60	200～300	900～1000

微软官方也放出使用高性能异步方法setImmediate与原始setTimeout的对比实验，流畅度很好，如图14-3所示。

```
http://ie.microsoft.com/testdrive/Performance/setImmediateSorting/Default.html
```

在现实中，尤其是游戏中，结合多种异步API是很有必要的。比如作为背景的树木、流水，NPC用requestAnimationFrame实现；而玩家角色，由于需要点击，再配合速度、体力、耐力等元素，其走路的速度是可变的，所以用setTimeout比较合适。一些非常炫的动画，可能就需要postMessage、Image.onerror、setImmediate、MessageChannel等API了。

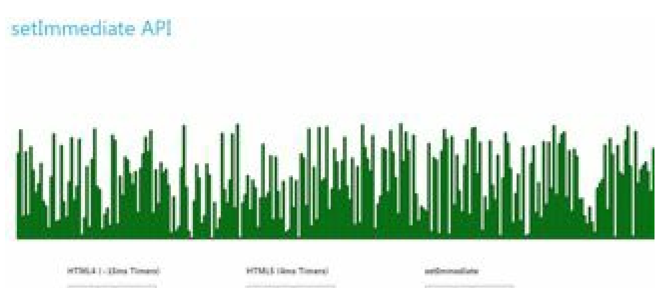


图14-3

14.6　CSS3 transition

transition是CSS3的一个重要模块，是CSS对入侵行为层的主要行为。W3C标准中对它是这样描述的：css 的transition 允许css 的属性值在一定的时间区间内平滑地过渡。这种效果可以在鼠标单击、获得焦点、被点击或对元素任何改变中触发，并圆滑地以动画效果改变css 的属性值。

transition主要包含4个属性值：transition-property，样式名；transition-duration，持续时间；transition-timing-function，缓动公式：transition-delay，延迟多长时间才触发。下面分别来看这4个属性值。

1．transition-property

transition-property是用来指定当元素其中一个属性改变时执行transition效果，主要有以下几个

值：none（没有属性改变）；all（所有属性改变），这个也是其默认值；indent（元素属性名）。当其值为none时，transition马上停止执行；当指定为all时，元素产生任何属性值变化时都将执行transition效果；ident是可以指定元素的某一个属性值。其对应的类型如下。

（1）与颜色相关的样式，如background-color、border-color、color、outline-color等。

（2）与盒子模块、字体大小、间距、行高有关的样式，如word-spacing、width、vertical-align、top、right、bottom、left、padding、outline-width、margin、min-width、min-height、max-width、max-height、line-height、height、border-width、border-spacing、background-position等。

（3）透明度，opacity。

（4）变形相关，即transform 样式。

（5）阴影，如text-shadow、box-shadow。

（6）线性渐变与径向渐变，用于背景色径向渐变，如-webkit-gradient、-ms-linear-gradient，各浏览器间的差异不是一般的大。

2．transition-duration

动画持续时间，单位可以是s，也可以是ms。我们可以连续写两个持续时间，对应两个不同的样式的变换。如：

```
transition-duration: 6s

transition-duration: 120ms

transition-duration: 1s, 15s

transition-duration: 10s, 30s, 230ms
```

transition-duration: inherit

3．transition-timing-function

缓动公式，根据时间的推进去改变属性值的变换速率。它有6个可能的值。

（1）ease：（逐渐变慢）默认值，ease函数等同于贝塞尔曲线（0.25, 0.1, 0.25, 1.0）。

（2）linear：（匀速），linear函数等同于贝塞尔曲线（0.0, 0.0, 1.0, 1.0）。

（3）ease-in：（加速），ease-in函数等同于贝塞尔曲线（0.42, 0, 1.0, 1.0）。

（4）ease-out：（减速），ease-out函数等同于贝塞尔曲线（0, 0, 0.58, 1.0）。

（5）ease-in-out：（加速然后减速），ease-in-out函数等同于贝塞尔曲线（0.42, 0, 0.58, 1.0）。

（6）cubic-bezier：（该值允许你去自定义一个时间曲线），特定的cubic-bezier曲线。（x_1 , y_1 , x_2 , y_2 ）4个值特定于曲线上点P_1 和点P_2 。所有值需在［0, 1］区域内，否则无效。

其中cubic-bezier为通过贝赛尔曲线来计算“转换”过程中的属性值，如图14-4所示，初始默认值为default，通过改变P_1 （x_1 , y_1 ）和P_2 （x_2 , y_2 ）的坐标可以改变整个过程的Output Percentage。

其他几个属性的示意如图14-5所示。

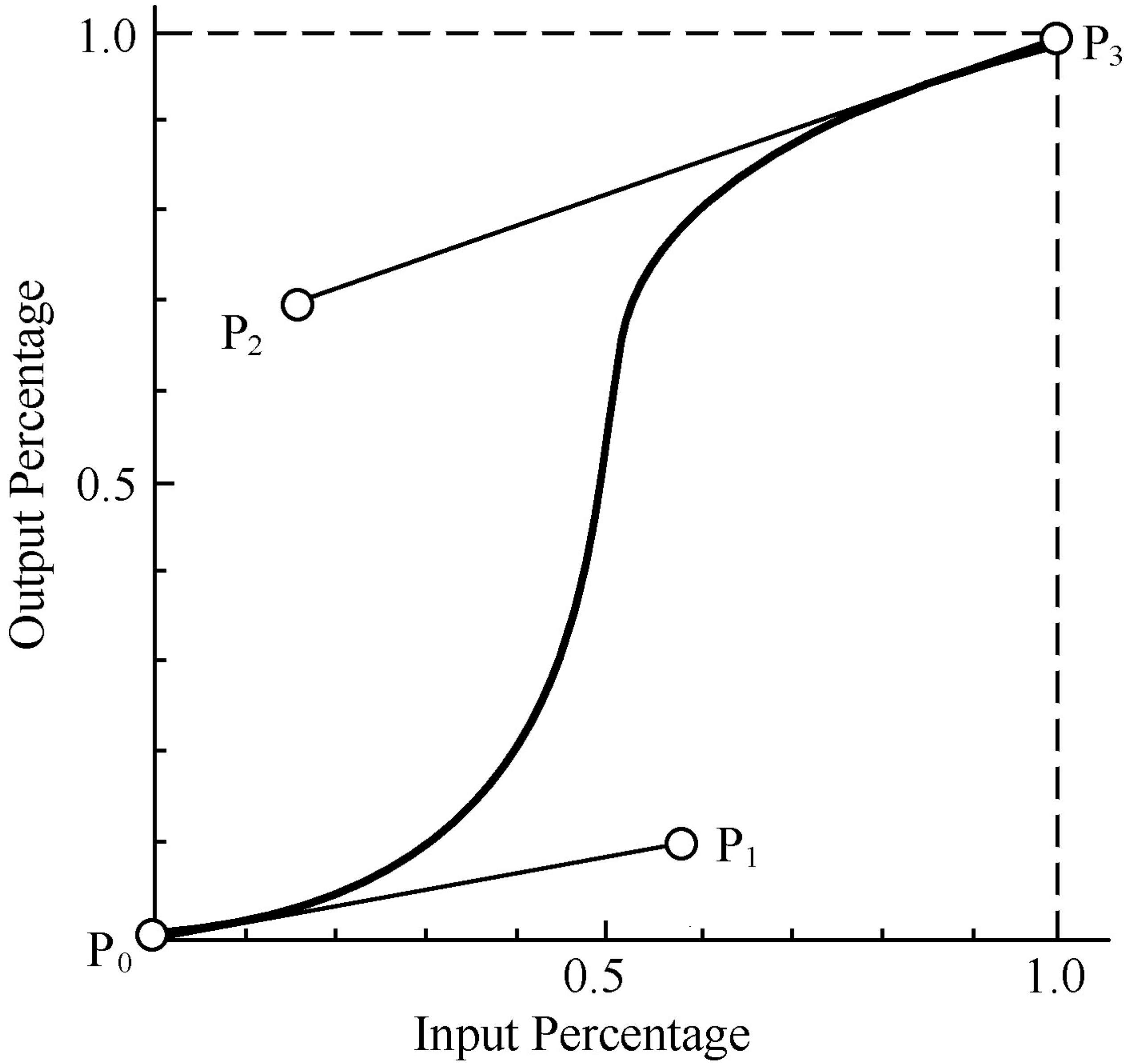


图14-4

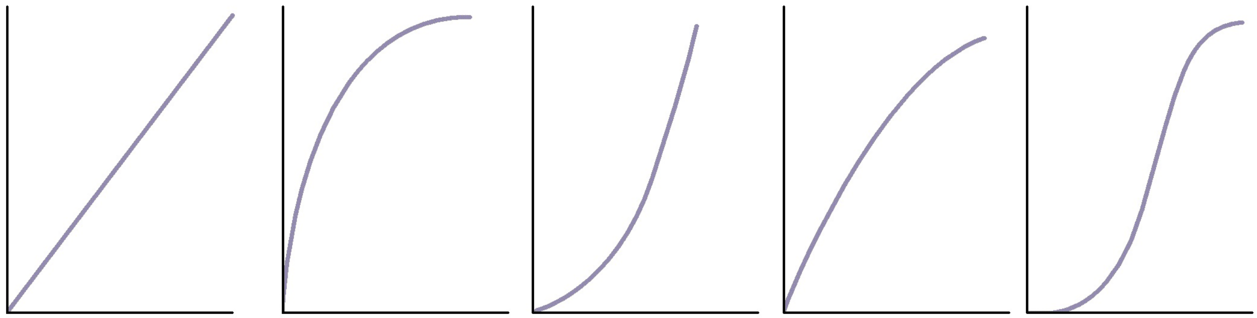


图14.5

4. transition-delay

延迟执行时间，可选单位有s与ms。

接着我们看如何应用。它必须是放在基于某些延迟触发的伪类或后来才添加到元素上的类名才有效。原因很简单，就是区分出初始状态与结束状态。比如一个元素的背景色原来是绿色，然后动态加了个类名或在hover中将它变成红色，这样transition才有用武之地。

```
<div id="move">移上去试试</div>
<style>
    #move {
        position: absolute;
        left:0px;
```

```
        width:100px;
        height:100px;
        background:red;
        font-size: 14px;
    }

    #move:hover {
        background:green;
        font-size: 26px;
        left:700px;
        -moz-transition: all 2s ease 0.3s;
        -webkit-transition: all 2s ease 0.3s;
        -o-transition: all 2s ease 0.3s;
        transition: all 2s ease 0.3s;
    }
</style>
```

它的支持情况如下，基本上现在我们都可以去掉私有前缀使用，如图14-6所示。

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	1.0 -webkit 26.0	4.0 (2.0) -moz 16.0 (16.0)	10.0	11.6 -o 12.10	3.0 -webkit

图14-6

另外，浏览器还提供了一个动画结束事件给我们监听。著名的Bootstrap的动画就是基于transition的。它的动画都是很简单的淡入淡出。另一个移动库zepto提供了更为强大的动画封装。

动画结束事件的麻烦在于，它的名字在各浏览器内严重不一致，在Opera竟然存在3种写法，如图14-7所示。

Feature	Chrome	Firefox (Gecko)	IE	Opera	Safari (WebKit)
1.0 -webkit 26.0	4.0 (2.0) -moz 16.0 (16.0)	10	10.5 -o 12.10	3.2 -webkit	
event name	1.0 as webkitTransitionEnd 26.0	4.0 (2.0)	10	10.5 as oTransitionEnd 12 as otransitionend 12.10 as transitionend	3.2 as webkitTransitionEnd

图14-7

那么如何精确取得可用的事件名呢？浏览器在全局作用域下暴露了许多事件的构造器，比如MouseEvent、MessageEvent、KeyboardEvent、UIEvent、MutationEvent、PopStateEvent、CloseEvent、StorageEvent、WheelEvent、WebKitTransitionEvent、WebKitAnimationEvent等。将这些构造器的名字直接传入到createEvent这个最底层的API中，如果不抛错，就说明支持这种事件。我们将此事件与对应类型组成个表，循环一下就能得到可用事件名了。

```
var getTransitionEndEventName = function() {
```

```
    var obj = {
        'TransitionEvent': 'transitionend',
        'WebKitTransitionEvent': 'webkitTransitionEnd',
        'OTransitionEvent': 'oTransitionEnd',
        'otransitionEvent': 'otransitionEnd'
    }
    var ret
    //有的浏览器同时支持私有实现与标准写法，比如webkit支持前2种，Opera支持1、3、4
for (var name in obj) {
        if(window[name]){
            ret = obj[name]
            break;
       }
        try {
            var a = document.createEvent(name);
            ret = obj[name]
            break;
        } catch (e) {
        }
    }//这是一个惰性函数，只检测一次，下次直接返回缓存结果
    getTransitionEndEventName = function() {
        return ret
    }
    return ret
}
alert(getTransitionEndEventName());
```

我们用JavaScript让一个元素动起来，然后再让它停下来。

```
<div id="test"></div>
<br/><br/><br/><br/><br/>
<button id="run">开始动画</button>
<button id="stop">中断动画并移除transition效果</button>
<button id="invalid">这次赋值将不会出现动画</button>
<style>
    #test{
```

```
        width:100px;
        height:100px;
        position: absolute;
        background: red;
        left:0;
        -moz-transition: left 5s;
        -o-transition:left 5s;
        -webkit-transition:left 5s;
        transition:left 5s;
    }
</style>
<script>
    var $ = function(a) {
        return document.getElementById(a)
    }
    var el = $("test")
    $("run").onclick = function() {
        el.style.left = "700px"
    }
    $("stop").onclick = function() {
        var left = window.getComputedStyle(el, null).left;
        el.style.left = left; //暂停
        ["", "-moz-", "-o-", "-webkit-"].forEach(function(prefi
x) {
            el.style.removeProperty(prefix + "transition")
        })
    }
    $("invalid").onclick = function() {
        el.style.left = "340px";
    }
</script>
```

上面的实验暴露出transition的弱点了，虽然暂停时我们是通过取当时值再重赋的手段实现了，但只要transition这个样式没有被清除掉，那么每一次变动left这个样式值都会发生动画。而想移除

transition这个样式，唯有它是写在标签时，我们才能通过removeProperty的方法实现，如果它是定义于内部样式或外部样式，那就不好处理了。如果外部样式还是跨域了，那么删除样式规则的这一手段也失灵了。因此这动画的transition可控度太差，不适宜作为一个框架的动画引擎的实现手段。

14.7 CSS3 animation

animation是CSS3另一个重要的模块，它成形得比transition晚，吸引了Flash的关键帧的理念，并克服了transition的一些缺陷，实用性非常高。animation是一个复合样式，它可以细分为8个更细的样式，情况与background和background-color、background-position、background-repeat、background-attachment、background-image的关系相仿。

1．animation-name

它所制约的是关键帧样式规则的名字，关键帧样式规则也就是以@keyframes开头的样式规则。animation-name可以同时对应多个关键帧样式规则名，以“，”号分开，说明此样式规则对多个关键帧样式规则都有效。

2．animation-duration

动画持续时间，单位是为s或ms，与transition相仿。

3．animation-timing-function

缓动公式，请参看transition同名项目的讲解。

4．animation-delay

动画延迟多久才开始，此时间不计入animation-duration。

5．animation-iteration-count

动画播放次数，值可以为正整数或infinite，默认只执行一次。这是一个很好的设计，不像transition，只要对指定样式重新赋值，就会触发意外的动画执行。

6．animation-direction

动画执行的方向，有4个值：normal、alternate、reverse、alternate-reverse。normal是指每次都是从第1帧（@keyframes中0%或from，不写，浏览器会自动补上）开始。alternate，这个值显然在 animation-iteration-count的值大于1 时才有效，它是指动画像钟摆一样从0%～100%，下次再从100%～0%，再一次又是0%～100%……reverse，这个有兼容问题，animation最早是由Safari发明的CSS样式，它最早制定的规范中并没有reverse这个值，它的行为与normal相反，每次都是从100%开始。alternate-reverse，也有兼容性问题，行为也是一个钟摆，第一次从100%摇到0%，下次是0%～100%……

7．animation-fill-mode

指动画跑完一圈（从0%～100%或从100%～0%）后，是保持动画前的状态forwards，还是此时的状态backwards。

8．animation-play-state

用于暂停（paused）或继续（running）此动画。

除了最后2个，前6个可以连写在一块，如图14-8所示。

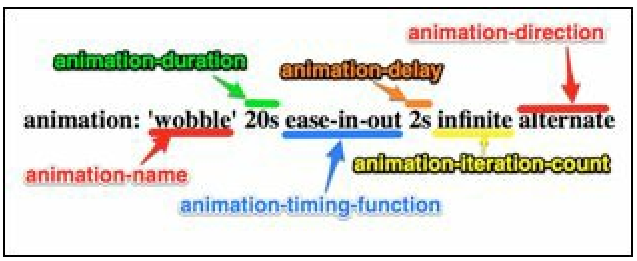


图14-8

此外，animation还配套了3种事件，分别用于开始时（animationstart）、结束时（animationend）、重复播放时（animationiteration）。当然，这些事件名可能不能直接用，需要加上私有前缀。基于上一节对transitionend取名字的经验，这也不是什么难事。

```
var getAnimationEndEventName = function() {
    //大致上有两种选择
    //IE10+, Firefox 16+ & Opera 12.1+: animationend
    //Chrome/Safari: webkitAnimationEnd
    //http://blogs.msdn.com/b/davrous/archive/2011/12/06/introd
uction-to-css3-animations.aspx
    //IE10也可以使用MSAnimationEnd监听，但是回调里的事件 type依然为ani
mationend
    //  el.addEventListener("MSAnimationEnd", function(e) {
    //     alert(e.type)// animationend！！！
    // })
    var obj = {
        'AnimationEvent': 'animationend',
        'WebKitAnimationEvent': 'webkitAnimationEnd'
    }
    var ret ;
    for (var name in obj) {
        if(window[name]){
            ret = obj[name];
            break;
        }
    }//这是一个惰性函数，只检测一次，下次直接返回缓存结果
    getAnimationEndEventName = function() {
        return ret;
    }
    return ret;
}
```

以下代码可以不断将一个正方体变成圆形，然后圆形再变成正方形，周而复始……

```
<div id="test"></div>
<style>
    #test{
        width:100px;
        height:100px;
        background: red;
        -webkit-animation: circle 1s infinite alternate;
```

```
        animation: circle 1s infinite  alternate;

    }
    @-webkit-keyframes circle{
        100%{
        -webkit-border-radius: 50px;
    }
    }
    @keyframes circle{
        100%{
        border-radius: 50px;
    }
    }
</style>
```

14.8 mass Framework基于CSS的动画引擎

本章最后一节，我们尝试利用CSS3 animation做个动画引擎。基于浏览器的动画API，性能比较高，尤其在移动端，它的优势就更明显了。由于animation在IE10才支持，因此如果想应用于PC端，自己要做一下适配。如果条件不满足，则退回基于JavaScript的动画引擎。如果我们的框架是基于AMD，这个实现起来就很简单了。

首先，我们看一下判定条件，方便切换。上面说过，浏览器把所有事件类型的构造器都放在window，只不过不可遍历，我们用Object.getOwnPropertyNames加filter一下子就能得到所有事件构造器。不难看出，只要存在window.AnimationEvent 或winodw.WebKitAnimationEvent 就可以使用我们基

于CSS的动画引擎。另一个方法是判定有没有keyframe样式规则的构造器，它也是放在window上，我们利用短路或把它所有可能的名字都放在一起就能判定出来。

```
var ok = window.MozCSSKeyframeRule || window.WebKitCSSKeyframeR
ule || window.CSSKeyframeRule;
```

用CSS实现动画引擎，有几个好处：它自带了缓动参数给你用；不用你计算原始值，它自行内部计算；颜色值不用你转换为RBG数组；如果想做倒带动画，那么直接设置animation- iteration-count为2，animation-direction为alternate就行了；像hide这个特效需要我们在动画结束时，将原来的进行动画的样式还原为初始值，在CSS3，我们只需animation-fill-mode设置为backwards；至于暂停与继续，其实就是控制animation-play-state的事。

与JavaScript动画引擎相比，CSS动画引擎在操

作元素进行动画时是通过添加类名与插入样式规则实现的。由于已进入IE10时代，我们可以直接使用el.classList.add来添加类名，动态插入样式可以有点偏门，但在支持animation的浏览器中，相关API已经没有兼容性问题了。

在浏览器中，有两个元素能生产样式表，link与style。我们可以通过访问其sheet来访问其样式表对象，然后在它的下面有个CSSRules类数组对象，里面就包含所有样式规则。为了方便操作，我们把动画引擎自己产生的样式规则全部放到一个动态插入的style元素中，以后删除就在这个元素里面找，这样可以减少一重遍历。而样式规则至少有5种类型，看以下代码。

```
.move {
    animation: move 4s linear;
}
keyframes move {
     from { margin-left:-20%; }
     to { margin-left:100%; }
}
font-face {
     font-family:'YourWebFontName';
```

```
     src:url('YourWebFontName.eot?') format('eot');/*IE*/
     src:url('YourWebFontName.woff') format('woff'), url('YourWebFontName.ttf') format ('truetype');/*non-IE*/
}
@media screen {
     #element { background:lightgreen;
}
```

从上到下依次是CSSStyleRule、CSSKeyframesRule、CSSFontFaceRule、CSSMediaRule。别忘了，CSSKeyframesRule还镶嵌着以百分比命名的CSSKeyframeRule。CSSStyleRule是最早的类型，我们可以通过其selectorText取得指定的样式规则，比如这里selectorText为.move。CSSKeyframes- Rule就是以@keyframes（视浏览器也可能是@-webkit-keyframe、@-moz-keyframe）开头的样式规则，我们可以通过专有的name属性判定。它里面指定进度该呈现的样式规则，用户在定义时可能用到to、from，但到 DOM 时全部转换为百分比了，它们可以通过keyText属性进行区分。CSSFontFaceRule用

于加载自定义字体，CSSMediaRule则应用于著名的响应式布局，这两个都没有什么名字可区分。不过没关系，我们只用到最前面两个。

下面mass Framework用于操作样式规则的方法。

```
https://github.com/RubyLouvre/mass-Framework/blob/master/fx_neo
.js
var styleElement;
function insertCSSRule(rule) {
    //动态插入一条样式规则
    if (styleElement) {
        var number = 0;
        try {
            var sheet = styleElement.sheet;// styleElement.styl
e Sheet;
            var cssRules = sheet.cssRules; // sheet.rules;
            number = cssRules.length;
            sheet.insertRule(rule, number);
        } catch (e) {
            $.log(e.message + rule);
        }
    } else {
        styleElement = document.createElement("style");
        styleElement.innerHTML = rule;
        document.head.appendChild(styleElement);
    }
}

function deleteCSSRule(ruleName, keyframes) {
    //删除一条样式规则
    var prop = keyframes ? "name" : "selectorText";
    var name = keyframes ? "@keyframes " : "cssRule ";//调试用
    if (styleElement) {
        var sheet = styleElement.sheet;// styleElement.styleShe
```

```
et;
        var cssRules = sheet.cssRules;// sheet.rules;
        for (var i = 0, n = cssRules.length; i < n; i++) {
            var rule = cssRules[i];
            if (rule[prop] === ruleName) {
                sheet.deleteRule(i);
                $.log("已经成功删除" + name + " " + ruleName);
                break;
            }
        }
    }
}

function deleteKeyFrames(name) {
    //删除一条@keyframes样式规则
    deleteCSSRule(name, true);
}
```

接着是引擎的主函数，$.fn.animtate，我们要设计与jQuery保持一致，降低学习成本，也意味着需要花许多代码处理参数多态化。还要考虑如何实现排队。以前我们是放在一个中央队列中，每一个元素都是一个很复杂的对象，用于分解成关键帧。现在我们没有必要搞队列了，我们可以在元素的animationend回调中自动执行下一个动画，所有排队的动画全部放到元素对应的缓存体中就行了。

```
function addOption(opts, p) {
    switch (typeof p) {
```

```
        case "object":
            $.mix(opts, p);
            delete p.props;
            break;
        case "number":
            opts.duration = p;
            break;
        case "string":
            opts.easing = p;
            break;
        case "function":
            opts.complete = p;
            break;
    }
}

function addOptions(duration) {
    //这里与JavaScript动画引擎大同小异
    var opts = {};
    for (var i = 1; i < arguments.length; i++) {
        addOption(opts, arguments[i]);
    }
    duration = opts.duration;
    duration = /^\d+(ms|s)?$/.test(duration) ? duration + "" :
"1000ms";
    if (duration.indexOf("s") === -1) {
        duration += "ms";
    }
    opts.duration = duration;
    opts.effect = opts.effect || "fx";
    opts.queue = !!(opts.queue == null || opts.queue); //默认使用
队列
    opts.easing = easingMap[opts.easing] ? opts.easing : "easeI
n";
    return opts;
}
```

上面用到一个easingMap对象，里面包含Robert Penner整理的所有缓动公式名及其对应的贝塞尔曲

线实现。由于CSS动画引擎体积很少，我们有足够空间将它们全部打包。

```
var easingMap = {
    "linear": [0.250, 0.250, 0.750, 0.750],
    "ease": [0.250, 0.100, 0.250, 1.000],
    "easeIn": [0.420, 0.000, 1.000, 1.000],
    "easeOut": [0.000, 0.000, 0.580, 1.000],
    "easeInOut": [0.420, 0.000, 0.580, 1.000],
    //……略……
    "custom": [0.000, 0.350, 0.500, 1.300],
    "random": [Math.random().toFixed(3),
        Math.random().toFixed(3),
        Math.random().toFixed(3),
        Math.random().toFixed(3)]
}
```

此外，外国还有些网站提供可视化的界面让你自己设计喜欢的曲线。

接着就是3个重要的内容函数：startAnimation、nextAnimation与stopAnimation。

startAnimation，用于立即执行此元素的动画，具体做法是分解原始材料构建两个样式规则，一个是用于集中定义动画的运作情况，另一个是定义第

一帧与最后一帧的样式情况。第一个样式规则是普通的CSSStyleRule，selectorText为一个类名，方便添加到目标元素上，另一个不用说是CSSKeyframesRule。由于是多个元素共用此类名，如果样式表有此类名，我们就不用重复分解与插入了。这里我们可以做个flag！最后我们要绑定一下animationend事件，在它的回调中保存指定样式到元素的style中，然后移除类名（因为类名对应的样式规则迟早会被移除，因此必须将它们转移到内联样式里），并调用nextAnimation与stopAnimation。

NextAnimation，决定是否调用startAnimation，里面有个setTimeout，用于模拟delay效果。

stopAnimation，用于移除startAnimation插入的两个样式规则。

```
var AnimationRegister = {};
function startAnimation(node, id, props, opts) {
    var effectName = opts.effect;
    var className = "fx_" + effectName + "_" + id;
    var frameName = "keyframe_" + effectName + "_" + id;
```

```
    var hidden = $.css(node, "display") === "none";
    var preproccess = AnimationPreproccess[effectName];
    if (typeof preproccess === "function") {
        var ret = preproccess(node, hidden, props, opts);
        if (ret === false) {
            return;
        }
    }
    //各种回调
    var after = opts.after || $.noop;
    var before = opts.before || $.noop;
    var complete = opts.complete || $.noop;
    var from = [],
            to = [];
    var count = AnimationRegister[className];
    node[className] = props;//保存到元素上，方便stop方法调用
    //让一组元素共用同一个类名
    if (!count) {
        //如果样式表中不存在这两条样式规则
        count = AnimationRegister[className] = 0;
        $.each(props, function(key, val) {
            var selector = key.replace(/[A-Z]/g, function(a) {
                return "-" + a.toLowerCase();
            });
            var parts;
            //处理show、toggle、hide 3个特殊值
            if (val === "toggle") {
                val = hidden ? "show" : "hide";
            }
            if (val === "show") {
                from.push(selector + ":0" + ($.cssNumber[key] ?
 "" : "px"));
            } else if (val === "hide") { //hide
                to.push(selector + ":0" + ($.cssNumber[key] ? "
" : "px"));
            } else if (parts = rfxnum.exec(val)) {
                var delta = parseFloat(parts[2]);
                var unit = $.cssNumber[key] ? "" : (parts[3] ||
 "px");
                if (parts[1]) { //操作符
                    var operator = parts[1].charAt(0);
                    var init = parseFloat($.css(node, key));
                    try {
                        delta = eval(init + operator + delta);
                    } catch (e) {
```

```
                        $.error("使用-=/+=进行递增递减操作时,单位只能为px, deg", TypeError);
                    }
                }
                to.push(selector + ":" + delta + unit);
            } else {
                to.push(selector + ":" + val);
            }
        });
        var easing = "cubic-bezier( " + easingMap[opts.easing] + " )";
        //CSSStyleRule的模板
        var classRule = ".#{className}{ #{prefix}animation: #{frameName} #{duration} #{easing} " +"#{count} #{direction}; #{prefix}animation-fill-mode:#{mode}  }";
        //CSSKeyframesRule的模板
        var frameRule = "@#{prefix}keyframes #{frameName}{ 0%{ #{from}; } 100%{  #{to}; }  }";
        var mode = effectName === "hide" ? "backwards" : "forwards";
        //填空数据
        var rule1 = $.format(classRule, {
            className: className,
            duration: opts.duration,
            easing: easing,
            frameName: frameName,
            mode: mode,
            prefix: prefixCSS,
            count: opts.revert ? 2 : 1,
            direction: opts.revert ? "alternate" : ""
        });
        var rule2 = $.format(frameRule, {
            frameName: frameName,
            prefix: prefixCSS,
            from: from.join("; "),
            to: to.join(";")
        });
        insertCSSRule(rule1);
        insertCSSRule(rule2);
    }
    AnimationRegister[className] = count + 1;
    $.bind(node, animationend, function fn(event) {
        $.unbind(this, event.type, fn);
        var styles = window.getComputedStyle(node, null);
        // 保存最后的样式
```

```
        for (var i in props) {
            if (props.hasOwnProperty(i)) {
                node.style[i] = styles[i];
            }
        }
        node.classList.remove(className); //移除类名
        stopAnimation(className);         //尝试移除keyframe
        after(node);
        complete(node);
        var queue = $._data(node, "fxQueue");
        if (opts.queue && queue) { //如果在列状,那么开始下一个动画
            queue.busy = 0;
            nextAnimation(node, queue);
        }
    });
    before(node);
    node.classList.add(className);
}
```

这里面有两个flag，AnimationRegister 里面装着许多类名，类名的值为数字，表示有多少个元素在共用它。我们只有在这个值为零时进行分解与插入样式规则。然后每当动画结束时，这个值就减一，归零时我们就移除它们。第二个flag是缓存体中的动画队列中busy，进行动画或被延迟时为真值，其他时间为假值，我们只有在假时，才能进入执行startAnimation的分支。

在startAnimation里，有时还会调用AnimationPreproccess里面的预处理函数，因此CSS3规定display为none的元素无法进行动画，因此我们想实现show特效，需提前修改display值。hide特效也要求我们对overflow做些处理。总体上说比JavaScript动画引擎简单多了。

nextAnimation与stopAnimation的源码如下。

```
function nextAnimation(node, queue) {
    if (!queue.busy) {
        queue.busy = 1;
        var args = queue.shift();
        if (isFinite(args)) {//如果是数字
            setTimeout(function() {
                queue.busy = 0;
                nextAnimation(node, queue);
            }, args);
        } else if (Array.isArray(args)) {
            startAnimation(node, args[0], args[1], args[2]);
        } else {
            queue.busy = 0;
        }
    }
}

function stopAnimation(className) {
    var count = AnimationRegister[className];
    if (count) {
        AnimationRegister[className] = count - 1;
        if (AnimationRegister[className] <= 0) {
            var frameName = className.replace("fx", "keyframe")
;
```

```
            deleteKeyFrames(frameName);
            deleteCSSRule("." + className);
        }
    }
}
```

最后，我们看看delay、pause、resume这几方法是如何实现的。

```
var playState = $.cssName("animation-play-state");
$.fn.delay = function(number) {
    return this.fx(number);
};
$.fn.pause = function() {
    return this.each(function() {
        this.style[playState] = "paused";
    });
};
$.fn.resume = function() {
    return this.each(function() {
        this.style[playState] = "running";
    });
};
```

一切就是这么简洁。

当然，基于CSS的动画引擎不是没有缺点，比如它对scrollTop、scrollLeft的动画就无能为力，它们是元素的属性。此外，我们也无法对canvas元素

里面的矢量图形进行动画。要想打包这一切，我们需要一个更强大的动画引擎。但canvas还涉及stage等概念，个人觉得还是针对它打包专用的引擎比较好。

有了基于CSS的动画引擎，我们就可以安心使用CSS3的transform2D或3D，要不用JavaScript来实现，在旧版本IE下连2D也很勉强。因为浏览器对元素进行变形，是基于矩阵，而不是rotate、scale、translate等原始函数。标准浏览器好办，getComputedStyle直接转换，IE需要自己提取参数值，转换角度为弧度，使用万能矩阵滤镜也惨兮兮的，另外，变形中心（transform-origin）也很难调校。如果两个连续的动画都涉及变形，那就是矩阵相乘。这里哗啦啦就要几百行，来实现矩阵的加减乘除……

矩阵相乘后，我们还得把这些值还原为

rotate、scale、translate、skew等方法的传参。在http://www.w3.org/TR/css3-transforms/这里，我们可以看到如何分解与还原矩阵的伪代码。显然，这一切加起来，让我们的引擎增加千行。这还不算transform3D。不过，它本来就无法在旧版本IE下实现。

在webkit与IE10中，各提供了一个CSSMatrix类（WebKitCSSMatrix与MSCSSMatrix），里面包含了我们所有想要的方法，否则自己实现有点残酷，如图14-9所示。

目前只是看IE11是否支持WebGL了，如果支持，到时肯定会提供更多的矩阵方法，我们的引擎玩3D才有意义。

最后奉上当今最出名的动画库一览表，如图14-10所示。

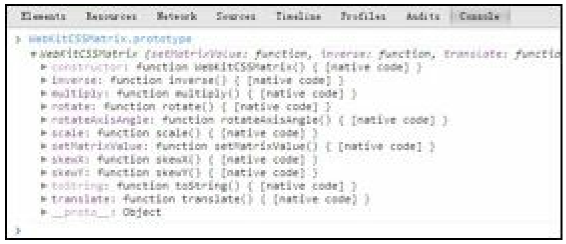
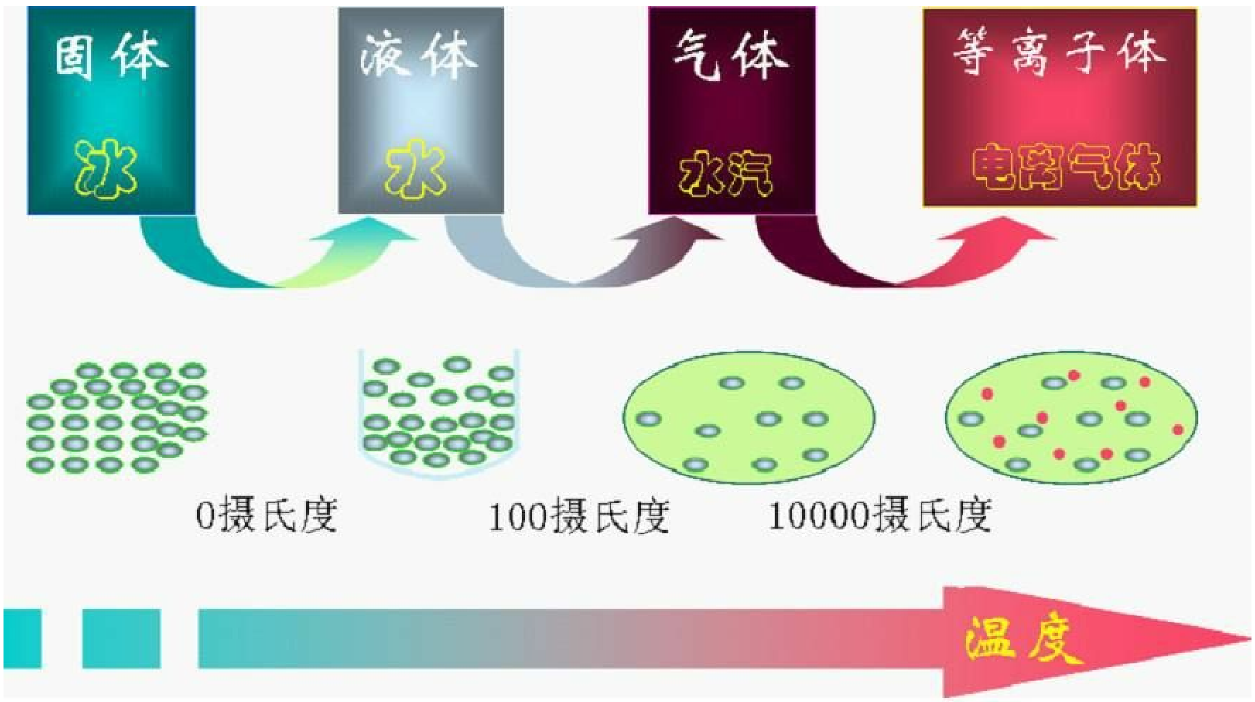

Elements
Resources
Network
Sources
Timeline
Profiles
Audits
Console
WebKitCSSMatrix.prototype
WebKitCSSMatrix {setMatrixValue: function, inverse: function, translate: functio
constructor: function WebKitCSSMatrix() { [native code] }
inverse: function inverse() { [native code] }
multiply: function multiply() { [native code] }
rotate: function rotate() { [native code] }
rotateAxisAngle: function rotateAxisAngle() { [native code] }
scale: function scale() { [native code] }
setMatrixValue: function setMatrixValue() { [native code] }
skewX: function skewX() { [native code] }
skewY: function skewY() { [native code] }
toString: function toString() { [native code] }
translate: function translate() { [native code] }
__proto__: Object

图14-9


MAP
Web Animation
Tools, Plugins & Libraries
Awwwards.com
JavaScript
SVG
Snap.js
Raphaël.js
SVG.js
Canvg
CSS3
Move.js
Animate.css
Effeckt.css
Animo.js
Zepto.js
Morf.js
GSAP
Jaguarjs
(Collie)
Tween.js
Motio
HTML5
Canvas
Fabric.js
bHive
Paper.js
kineticJS
Easel.js
jQuery
Plugins
WebGL
Three.js
Processing.js
Babylon.js
A) Animating DOM elements through CSS or JavaScript
B) Libraries to animate SVG Canvas WebGL

图14-10

第15章MVVM



人们总是爱探求完美的东西，编程界也有自己的追求，如完美的架构从MVC到MVP，再到MVVM。当然MVC、MVP、MVVM有它们不同的场景，但MVVM在微软试水后已被证实为界面开发最好的方案。本章将探索MVVM的一般实现方案。

15.1　前端模板（静态模板）

说起MVVM，就不得不说起其前身MVC与MVP，如图15-1所示。

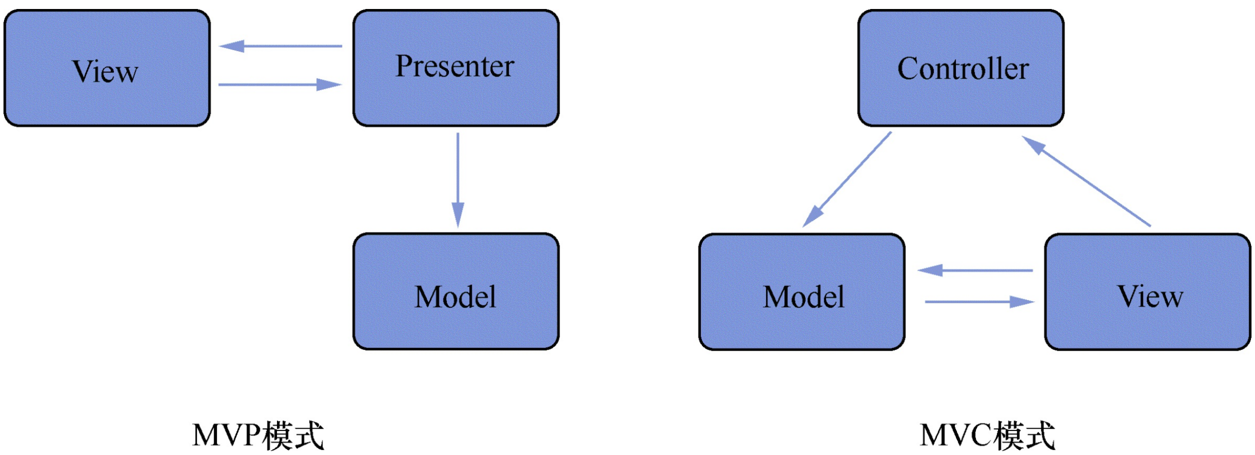


图15-1

MVC与MVP最大的区别是减少关注点，基本上只有Presenter在变动。但不管怎么样，前端框架最重要的目的是将页面渲染出来。“渲染”（render）这个词最初不是前端的东西。前端之前叫做切图，将设计师做的PSD变成一个静态页面，然后加上动态交互。但是我们有许多数据是来

自后端，如何将数据加入静态页面呢？于是又多了一套工序叫“套页面”。套页面的过程实际就是将静态页面切割成若干功能块，每一块都是一个php、jsp或vm文件，它们是后端模板引擎 的处理对象！其实模板是不局限于后端还是前端的，模板的本质是用于从数据（变量） 到实际的视觉表现（ **HTML** 代码） 这项工作的一种实现手段。由于后端近水楼台先得月（取数据比较方便），因此先在后端发展出这种技术。这些后端模板文件是活动于服务器的，然后经过复杂的处理，最后由浏览器渲染出来。这时的渲染是将服务器拼接好的静态文本变成一个DOM树的过程。

如果要前端实现MVC或MVP或MVVM，那些工序必须发生改变。静态文件产出是不变的，尤其是大公司，分工够细，有专门的切图组将它们做出来。接着是套页面，这时就不能使用后端模板引擎，需要引入前端模板引擎。由于实现一个前端模

板引擎太简单了，经过多年的发展，已经有众多好用的“轮子”。

```
https://github.com/janl/mustache.js
```

基于JavaScript的Logic-less（无逻辑或轻逻辑）模板。

```
https://github.com/twitter/hogan.js
```

上面的优化版，twitter出品。

```
https://github.com/wycats/handlebars.js
```

完全兼容mustcache的语法。

```
https://github.com/paularmstrong/swig
```

拥有更强悍的模板继承与block重写功能。

```
https://github.com/mozilla/nunjucks
```

跟django的模板系统相似，可以说swig的升级版，是gitbook的御用前端模板。

其他推荐的还有ejs，易学易用，对有过ASP/PHP/JSP编程经验的人来说，非常亲切自然，缺点就是功能有点简单。

其他的如doT、xtempalate、Underscore Templates。

最不推荐是jade，有点华而不实，过度设计，导致套页面工作量大，性能其差。

虚拟DOM时代流行的JSX就是无逻辑模板。之所以流行无逻辑或轻逻辑模板，其主要原因是改动成本比较少，像jade这样自造语法糖太多，从美工手中拿来的HTML需要大动干戈，进行摧心折骨般

的改造才能套数据。对于模板来说，最简单而言，就是将某个可变数据放到适当的地方（填空），而其次，可以控制这个区域输出不输入（**if** 指令），或让其个区域循环输入多次（**for** 指令），更强制，实现模板互相套嵌（**layout** 与 **block**）。为了实现if与for有两种方法，一种是单纯的区域，插入一个JS语句，里面有if语句与for语句，另一种是使用语法糖，比如说 ms-for、ms-repeat、ng-if、ng-repeat。语法糖的用法比直接使用JS语句简单，但是带来的是学习成本与拓展功能。每一个模板if、for指令的语法都不一样的，并且你想在循环做一些处理，比如过滤一些数据，或突然在某处中断，这又得引用一些新的语句。随着模板要求前后共用，就有了传输成本，直接写JS语句在模板里面肯定比不过语法糖。因此基于这种种原因，mustache风格的模板就成为主流。

现在3种模板风格。PHP/ASP/JSP风格。

```
<% if ( list.length ) { %>
  <ol>
    <% for ( n=0; n<list.length; ++n ) { %>
      <li>
        <%= list[n] %>
      </li>
    <% } %>
  </ol>
<% } %>
```

mustcache风格，高级语法有限，通常难自定义拓展。

```
{{#if list.length}}
  <ol>
    {{#each list item}}
      <li>
        {{ item }}
      </li>
    {{/each}}
  </ol>
{{/if}}
```

属性绑定风格。

```
<ol ms-if="list.length">
  <li ms-for="item in list">
      {{item}}
  </li>
</ol>
```

前两者只能出现于script、textarea等容器元素内部。因此<分隔符与标签的<容器造成冲突，并且也不利于IDE的格式化处理。属性绑定风格则是MVVM时期最流行的模板定义风格，某页面某个区域就是一个模板，不需要进行入append等操作。

我们再来看如何实现前端模板。前端模板的本质就是一个可以转换函数的字符串，这个函数放进一个充满数据的对象后，还原为一个全新的字符串。因此重点是如何构建一个渲染函数。最简单的方式是正则，还记得第二章的format方法吗，这就是一个轻型的填充数据的方法。

```
function format(str, object) {
    var array = Array.prototype.slice.call(arguments, 1);
    return str.replace(/\\?\#{([^{}]+)\}/gm, function(match, na
me) {
        if (match.charAt(0) == '\\')
            return match.slice(1);
        var index = Number(name)
        if (index >= 0)
            return array[index];
        if (object && object[name] !== void 0)
            return  object[name];
        return  '';
    });
}
```

format方法是通过#{ }来划分静态内容与动态内容的，一般来说它们称之为定界符（**delimiter**）。#{为前定界符，}为后界符，这个#{}其实是ruby风格的定界符。通常的定界符是<% 与%>，{{ 与}}。通常在前定界符中还有一些修饰符号，比如=号，表示这个会输出到页面，-号，表示会去掉两旁的空白。将下例，要编译成一个渲染函数。

```
var tpl = '你好,我的名字啊<%name%>, 今年已经 <%info.age%>岁了'
 var data = {
     name: "司徒正美",
     info: {
       age: 20
     }
 }
```

大抵是这样。

```
var body = '你好,我的名字叫'+ data.name+ ', 今年已经 '+data.info.age+ '岁了'
var render = new Function('data', 'return '+ body)
```

或者聪明一点，使用数组来join。

```
var array = ['return ']
array.push('你好,我的名字叫'）
array.push(data.name)
array.push(', 今年已经')
array.push(data.info.age)
array.push( '岁了')
var render = new Function('data', array.join('+'))
```

这就得区分静态内容与为变量前加data. 前缀。这一步可以用正则来做，也可以用纯字符串。我们试一下纯字符串方式。假令前定界符为openTag，后定界符为closeTag，通过indexOf 与slice方法，就可以将它切成一块块。

```
function tokenize(str) {
    var openTag = '<%'
    var closeTag = '%>'
    var ret = []
    do {
        var index = str.indexOf(openTag)
        index = index === -1 ? str.length : index
        var value = str.slice(0, index)
        //抽取{{前面的静态内容
        ret.push({
            expr: value,
```

```
            type: 'text'
        })
        //改变str字符串自身
        str = str.slice(index + openTag.length)
        if (str) {
            index = str.indexOf(closeTag)
            var value = str.slice(0, index)
            //抽取{{与}}的动态内容
            ret.push({
                expr: value.trim(),//JS逻辑两旁的空白可以省去
                type: 'js'
            })
            //改变str字符串自身
            str = str.slice(index + closeTag.length)
        }
    } while (str.length)
    return ret
}
console.log(tokenize(tpl))
```



然后通过render方法将它们拼接起来。

```
function render(str) {
     var tokens = tokenize(str)
     var ret = []
     for (var i = 0, token; token = tokens[i++]; ) {
         if (token.type === 'text') {
             ret.push('"' + token.expr + '"')
         } else {
             ret.push(token.expr)
         }
     }
     console.log("return "+ ret.join('+'))
 }
```

打印出来如下。

```
return "你好,我的名字叫"+name+", 今年已经 "+info.age+"岁了"
```

这个方法还不完整。首先只是在两旁加上双引号是不可靠的，万一里面还有双引号怎么办。因此我们需要引入第二章介绍的quote方法，当类型为文本时，`ret.push(+quote(token. expr)+)`。其次需要对动态部分的变量加上`.data`。怎么知道它是一个变量呢？我们回想一下变量的定义，就是以_、

$或字母开头的字符组合。为了简洁起见，我们暂时不用理会中文的情况。不过，`info.age`这个字符串里面，其实有两个符合变量的子串，而只需要在info前面加`data.`。这时，我们需要设法在匹配变量前，将对象的子级属性替换掉，替换成不符合变量的字符，然后再替换回去。为此，笔者搞了一个dig与fill方法，将子级属性变成`??12`这样的字符串。

```
var quote = JSON.stringify//自己到第二章找完整函数
var rident = /[$a-zA-Z_][$a-zA-Z0-9_]*/g
var rproperty = /\.\s*[\w\.\$]+/g
var number = 1
var rfill = /\?\?\d+/g
var stringPool = {}
function dig(a) {
    var key = '??' + number++
    stringPool[key] = a
    return key
}
function fill(a) {
    return stringPool[a]
}
function render(str) {
    stringPool = {}
    var tokens = tokenize(str)
    var ret = []
    for (var i = 0, token; token = tokens[i++]; ) {
        if (token.type === 'text') {
            ret.push(quote(token.expr))
        } else {
```

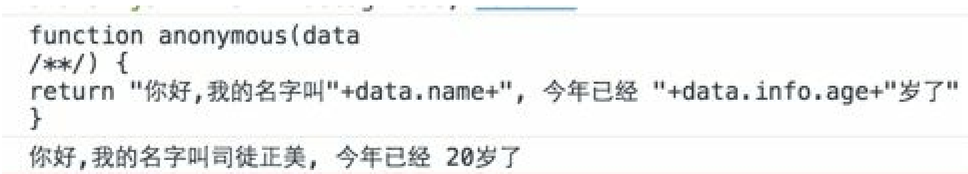
```
            // 先去掉对象的子级属性,减少干扰因素
            var js = token.expr.replace(rproperty, dig)
            js = js.replace(rident, function (a) {
                return 'data.' + a
            })
            js = js.replace(rfill, fill)
            ret.push(js)
        }
    }
    console.log("return " + ret.join('+'))
 }
render(tpl)
```

输出为如下代码。

```
return "你好,我的名字叫"+data.name+", 今年已经 "+data.info.age+"岁了"
```

最后，我们修改一下后面两行，得到我们梦魅以求的渲染函数，它的实现过程比format方法复杂多了，但却是所有扩展性极强的前端模板的一般实现过程。

```
  function render(str){
  //略……
     return new Function("data", "return " + ret.join('+'))
  }
  var fn = render(tpl)
  console.log(fn+"")
  console.log(fn(data))
```



```
function anonymous(data
/**/) {
return "你好,我的名字叫"+data.name+", 今年已经 "+data.info.age+"岁了"
}
你好,我的名字叫司徒正美, 今年已经 20岁了
```

我们再看一下如何引入循环语句，比如将上面的模板与数据改成这样。

```
var tpl = '你好,我的名字叫<%name%>, 今年已经 <%info.age%>岁了,喜欢<%
 for(var i = 0, el; el = list[i++];){%><% el %> <% } %>'
 var data = {
     name: "司徒正美",
     info: {
         age: 20
     },
     list: ["苹果","香蕉","雪梨"]
 }
```

这时我们就添加一种新的类型，不输出到页面的动态内容，这在token方法中做一些修改。

```
value = value.trim()
if (/^(if|for|})/.test(value)) {
    ret.push({
        expr: value,
        type: 'logic'
    })
} else {
    ret.push({
        expr: value,
```

```
        type: 'js'
    })
}
```

但render方法怎么修改好呢，显示这时继续用 + 已经不行了，否则下场是这样。

```
return "你好,我的名字叫"+data.name+", 今年已经 "+data.info.age+"岁了,喜欢"+for(var i = 0, el; el = list[i++];){+""+data.el+" "+}
```

我们需要借用数组，将要输入的数据（text, js 类型）放进去，logic类型不放进去。

```
function addPrefix(str) {
   // 先去掉对象的子级属性,减少干扰因素
   var js = str.replace(rproperty, dig)
   js = js.replace(rident, function (a) {
       return 'data.' + a
   })
   return js.replace(rfill, fill)
}
function addView(s) {
   return '__data__.push(' + s + ')'
}
function render(str) {
    stringPool = {}
    var tokens = tokenize(str)
    var ret = ['var __data__ = []']
    tokens.forEach(function(token){
       if (token.type === 'text') {
           ret.push(addView(quote(token.expr)))
       } else if (token.type === 'logic') {
```

```
            //逻辑部分都经过addPrefix方法处理
            ret.push(addDataPrefix(token.expr))
        } else {
            ret.push(addView(addPrefix(token.expr)))
        }
    })
    ret.push("return __data__.join('')")
    console.log( ret.join('\n'))
  }
var fn = render(tpl)
```

得到的内部结构是这样的，显然addPrefix方法出问题，我们应该过滤掉if、for等关键字与保留字。

```
var __data__ = []
__data__.push("你好,我的名字叫")
__data__.push(data.name)
__data__.push(", 今年已经 ")
__data__.push(data.info.age)
__data__.push("岁了,喜欢")
data.for(data.var data.i = 0, data.el; data.el = data.list[data
.i++]){
__data__.push("")
__data__.push(data.el)
__data__.push(" ")
}
return __data__.join('')
```

但即使我们处理掉关键字与保留字，对于中间生成i、el怎么区分呢？是区分不了的。于是目前有

两种方法，一是使用with，这时我们就不需要加data. 前缀。第二种引入新的语法，比如，前面是@ 就替换为data 。

先看第一种。

```
function render(str) {
     stringPool = {}
     var tokens = tokenize(str)
     var ret = ['var __data__ = [];', 'with(data){'}
     for (var i = 0, token; token = tokens[i++]; ) {
         if (token.type === 'text') {
             ret.push(addView(quote(token.expr)))
         } else if (token.type === 'logic') {
             ret.push(token.expr)
         } else {
             ret.push(addView(token.expr))
         }
     }
     ret.push(')')
     ret.push('return __data__.join("")')
     return new Function("data", ret.join('\n'))
 }
 var fn = render(tpl)
 console.log(fn + "")
 console.log(fn(data))
```

```
function anonymous(data
/**/) {
var __data__ = [];
with(data){
__data__.push("你好,我的名字叫")
__data__.push(name)
__data__.push(", 今年已经 ")
__data__.push(info.age)
__data__.push("岁了,喜欢")
for(var i = 0, el; el = list[i++];){
__data__.push("")
__data__.push(el)
__data__.push(" ")
}
}
return __data__.join("")
}
```

```
你好,我的名字叫司徒正美, 今年已经 20岁了,喜欢苹果 香蕉 雪梨
```

许多迷你模板都是用with减少替换工作。

第二种方法，使用引导符@，avalon2就是这么用的。这样addPrefix方法可以减少许多代码。相对应，模板也要改动一下。

```
var tpl = '你好,我的名字叫<%@name%>, 今年已经 <%@info.age%>岁了,喜欢<% for(var i = 0, el; el = @list[i++];){%><% el %> <% } %>'
var rguide = /(^|[^\w\u00c0-\uFFFF_])(@|##)(?=[$\w])/g
function addPrefix(str) {
    return str.replace(rguide, '$1data.')
}
function render(str) {
    stringPool = {}
    var tokens = tokenize(str)
    var ret = ['var __data__ = [];']
    for (var i = 0, token; token = tokens[i++]; ) {
        if (token.type === 'text') {
            ret.push(addView(quote(token.expr)))
        } else if (token.type === 'logic') {
            //逻辑部分都经过addPrefix方法处理
            ret.push(addPrefix(token.expr))
```

```
        } else {
            ret.push(addView(addPrefix(token.expr)))
        }
    }

    ret.push('return __data__.join("")')
    return new Function("data", ret.join('\n'))
}
var fn = render(tpl)
console.log(fn + "")
console.log(fn(data))
```

```
function anonymous(data
/**/) {
var __data__ = [];
__data__.push("你好,我的名字叫")
__data__.push(data.name)
__data__.push(", 今年已经 ")
__data__.push(data.info.age)
__data__.push("岁了,喜欢")
for(var i = 0, el; el = data.list[i++];){
__data__.push("")
__data__.push(el)
__data__.push(" ")
}
return __data__.join("")
}
你好,我的名字叫司徒正美, 今年已经 20岁了,喜欢苹果 香蕉 雪梨
```

第二种比第一种的优势在于，性能更高，并且避开ES5严格模式的限制。

我们再认真思考一下，其实循环语句与条件语句，不单是for、if两个，还有while、do while、else等。因此这需要优化，也有两种方法。第一种是，添加更多语法符合，比如上面所说的= 就是输出，

没有则不输出。这是ASP/JSP/PHP等模板采用的手段。

```
if (value.charAt(0) === '=') {
     ret.push({
         expr: value,
         type: 'js'
     })
 } else {
     ret.push({
         expr: value,
         type: 'logic'
     })
 }
```

另一种，使用语法糖，如`#each(el, index) in @list`，‘#eachEnd’，‘#if’，‘#ifEnd’。还是改动tokenize方法。

```
 if (value.charAt(0) === '#') {
    if (value === '#eachEnd' || value === '#ifEnd') {
        ret.push({
            expr: '}',
            type: 'logic'
        })
    } else if (value.slice(0, 4) === '#if ') {
        ret.push({
            expr: 'if(' + value.slice(4) + '){',
            type: 'logic'
        })
    } else if (value.slice(0, 6) === '#each ') {
        var arr = value.slice(6).split(' in ')
        var arrayName = arr[1]
```

```
        var args = arr[0].match(/[$\w_]+/g)
        var itemName = args.pop()
        var indexName = args.pop() || '$index'
        value = ['for(var ', ' = 0;', '<' + arrayName + '.lengt
h;', '++){'].join(indexName) +
                '\nvar ' + itemName + ' = ' + arrayName + '[' +
 indexName + '];'
        ret.push({
            expr: value,
            type: 'logic'
        })

    }

} else{
   //…
}
```

对应的模板改成如下代码。

```
var tpl = '你好,我的名字叫<%@name%>, 今年已经 <%@info.age%>岁了,喜欢
<%#each el in @list %><% el %> <% #eachEnd %>'
var fn = render(tpl)
console.log(fn + "")
console.log(fn(data))
```

```
function anonymous(data
/**/) {
var __data__ = [];
__data__.push("你好,我的名字叫")
__data__.push(data.name)
__data__.push(", 今年已经 ")
__data__.push(data.info.age)
__data__.push("岁了,喜欢")
for(var $index = 0;$index<data.list.length;$index++){
var el = data.list[$index];
__data__.push("")
__data__.push(el)
__data__.push(" ")
}
return __data__.join("")
}
```

```
你好,我的名字叫司徒正美, 今年已经 20岁了,喜欢苹果 香蕉 雪梨
```

可能有人觉#for、#forEnd这样的语法糖比较丑，没问题，这个可以改，主要我们的tokenize方法足够强大，就能实现mustache这样的模板引擎。但所有模板引擎也基本上是这么实现的，有的还支持过滤器，也就是在js 类型的语句再进行处理，将丨后面的字符器再切割出来。

如果虚拟DOM呢？那就需要一个html parser，这个工程巨大，比如reactive这个库，早期不使用html parser与虚拟DOM，只有3、4千行，加入这些炫酷功能后就达到16000行。返回一个字符串与返回一个类似DOM树的对象树结构是不一样的。

15.2　MVVM的动态模板

早期前端模板都是静态模板。静态模板都是返回一个字符串，然后通过innerHTML对目标区域进行铺天盖地式的大替换。

```
el.innerHTML = templateString
```

大家回忆一下第二章所说的innerHTML，这东西太多bug。因此我们最好用jQuery的html方法进行替换操作。但即便这样，静态模板的效率还是很低下，并且存在如下几个隐患。

- 破坏了对事件的绑定。
- 破坏了原区域的选区与光标。
- 破坏了原区域的第三方组件。

随着jQuery的地位日益巩固，大家可以安心用

jQuery作为公司的标准库，然后利空闲时间研究其他东西。这个时间冒出许多东西。比如说上面提到的众多前端模板，各种UI库，还有语言扩展库（underscore）。复杂的交互需要大量的JS，人们开始研究如何大规模组织前端工程，于是这时期最伟大的产物，require.js出现了。

其实require.js不站出来，许多一些我们现在不知道的库也在做同样的事。统一整个社区的JS文件的编写形式，他们管这叫做模块。每个模块都要声明自己所依赖的另外的JS文件，及将自己的东西提供给别人优雅地调用（重要一点是不污染全局作用域）。

与require.js一起成长的是Node.js，让更多后端加入。更多后端加入意味着更多经过培训的专业大脑加进来。许多有20年经验的程序员，把后端的那一套直接复制过来，于是有了一大堆JavaScript

MVC 框架。那时还流行一篇文章《12款优秀的JavaScript MVC框架评估》，介绍了当时的各种MVC框架，如图15-2所示。

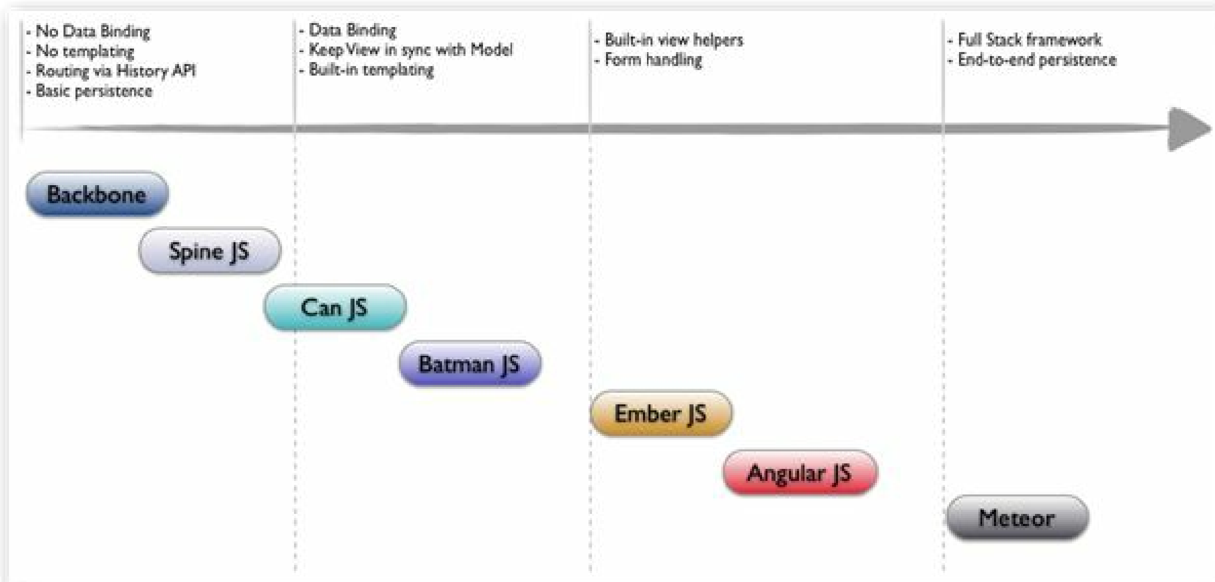


图15-2

这时knockout.js与angular就进入人们的眼帘了。Knockout.js是在2011年MIX11大会上发布的，算是历史最悠久的。angular则更晚些发布，但听说2009年就在内部使用了。但不管knockout还是angular，它们都有一个特点，需要页面添加许多特殊的标记。它们被官方称为指令或绑定属性。那片

区域其实就是MVVM的动态模板了。所有框架都在解决关注点分离，并且关注点越少越好。经典的3层架构，由展现层、业务逻辑层和持久层构成，其中体现了我们对用户界面、业务逻辑和数据持久的关注点分离。而MVVM架构，希望只有一个关注点，其他都是被动改变。MVVM中的VM，我们下一节详细介绍。现在回到动态模板上。动态模板是MVVM的视图层，它的特点是能最小化刷新。哪个变量发生改动，那么只有涉及变量的属性值，nodeValue才会改变。于是就没有原来静态模板的三大副作用，性能上也大为提升。

由于操作对象是属性值，nodeValue或单独几个要循环生成的元素节点。MVVM在页面刚完成渲染时，就会遍历整个DOM树，将它感兴趣的元素（即被指令标识的那些节点）全部收集起来。然后对指令进行解析，转换为求值函数，传入数据，得到当前值，再放进刷新函数，于是这个节点就

更新了！因此动态模板是没有产生一个完整的字符串或一片DOM，而是通过指令产生了许多细小的求值函数、刷新函数，通过关联这些函数与数据与节点的关系，实现视图更新的。这种关联方法就是观察者模式，当然还可以糅杂其他设计模式，但主要是观察者模式。因此我们可以说，动态模板就是观察者模式的架设！就像国家那样，通过电线杆与电线将农村连接在一起，如图15-3所示。

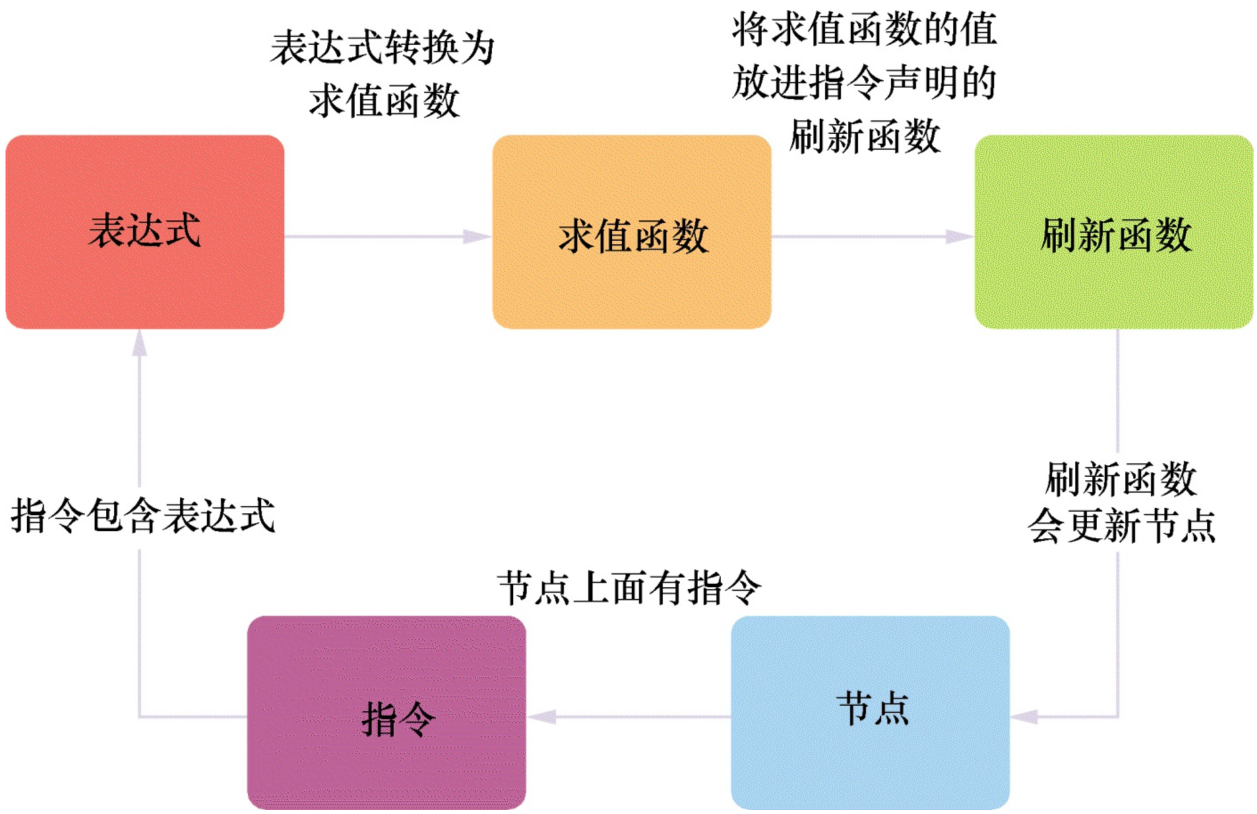


图15-3

```
var fn = parseExpr(binding.expr)
var value = fn(data)
var update = getUpdate(binding.type)
update(binding.element, value)
```

15.2.1　求值函数

如果大家用过avalon或angular，会看到ms-attr-title="el + 222"、ng-attr-title= "el+

222" 这样的东西。这就是指令。指令是什么呢？一个标记。标记它会对当前元素进行怎么样的操作呢？通常指令是以元素属性的形式存在，但angular存在多种指令形式，tagName、comment、attribute、className与双花括号形式的插值。MVVM框架通常有一个扫描过程，将所有它感兴趣的元素进行收集。这个扫描与收集后面会说。而这些它感兴趣的元素都是带有指令的。

以属性形式的指令为例，通常我们称之为绑定属性，它以等号为分隔，分为属性名与属性值两大部分。属性名又多以ms-、ng-、：开头，我们称之为前缀，它唯一的作用是告诉框架，它是绑定属性。紧接着是绑定属性的类型，它告诉框架，以后要使用什么刷新函数来加工或刷新这个节点。紧接着是额外的传参，这个是可选的。最后是属性值部分，它被称为表达式，有时候它还带有过滤器，基本上都是使用UNIX下的管道符风格。一个表达

式里面可能包含多个过滤器。因此一个绑定属性至少包括类型与表达式两部分 。所有指令都会转换为一个绑定对象 。

一个绑定对象的结构大致如图15-4所示。

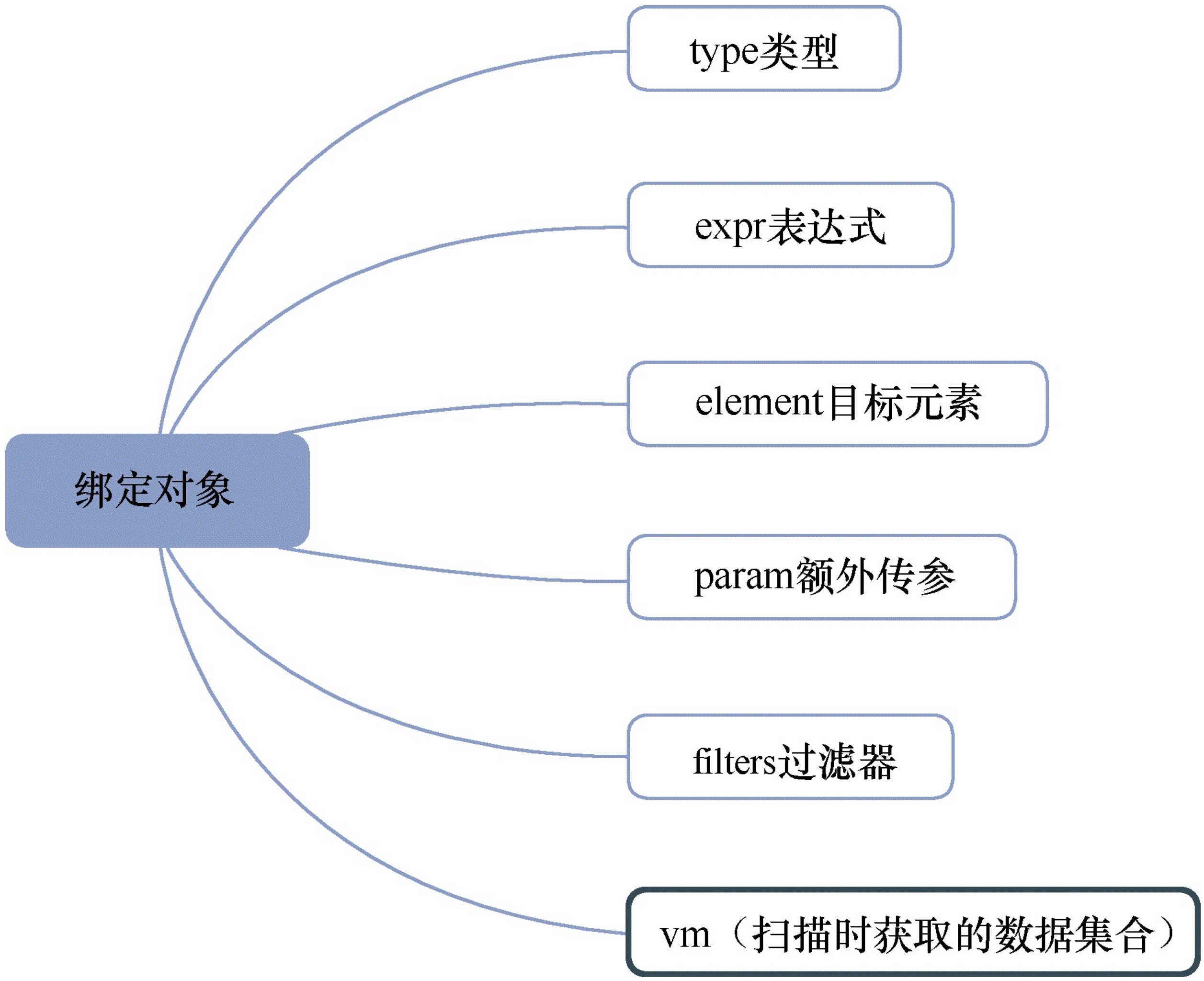


图15-4

执行流程是这样的，表达式会转换为求值函数（通常命名为**get**），传入数据与额外传参得到值，如果存在过滤器，那么还要依次放进这些过滤器中，最后将值与元素放进视图刷新函数中。并且这个值会保存下来，放到绑定对象上。当一次数据

发生变动时，会进行新值与旧值比较，决定它是否进入刷新函数这一步。

一个完整的绑定对象，它是在第一次被执行后获取其他额外成员，如图15-5所示。

操作流程大概如下。

```
if(!binding.get){
   binding.get = parseExpr(binding.expr, binding.vm)
}
var value = binding.get(binding.vm, binding.param)
if(binding.value !== value){
    binding.value = value
    if(binding.filter){
       value = binding.filter(value)
    }
    binding.update(binding.element, value)
}
```

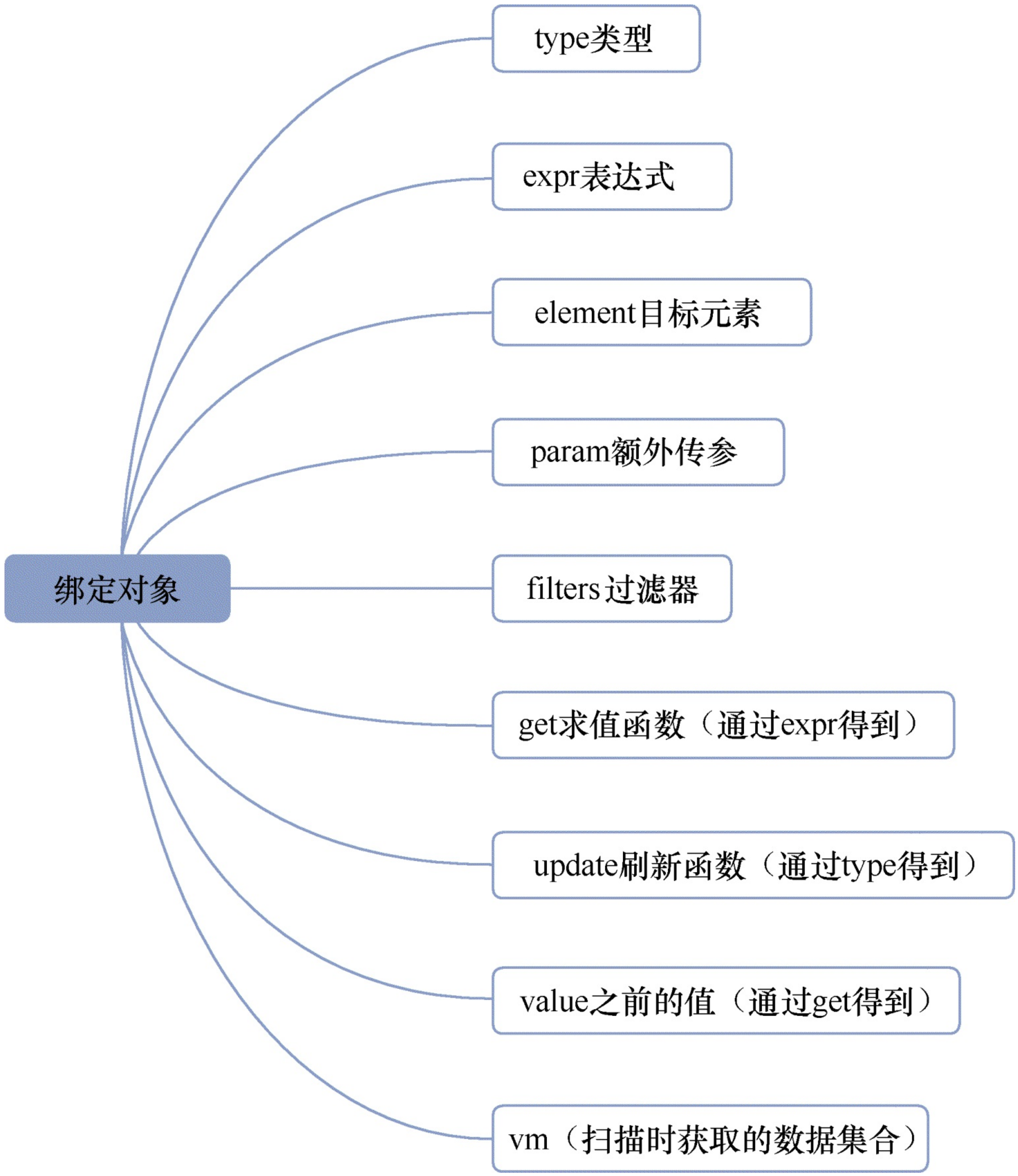


图15-5

通常来说刷新函数是框架内置的，有多少种绑

定属性就对应多少刷新函数。而求值函数则需要我们转换。这个转换与上节我们提到的前端模板转换为渲染函数很相似，都是将字符串变成函数。但如果从经典的MVVM角度出发，这个构建方式又有一点不一样。但不管怎么说，上次给出的转换函数确实不太完整。接着下来，我们搞一个完整的，能用于生产环境的方法。

将一个表达式变成一个函数，不是单单在前面加上一个“return”这么简单的。经典的MVVM是存在一个依赖收集的过程，需要触发访问器属性的getter方法，因此需要将变量前置。

```
el + 222
```

实际上会变成如下代码。

```
var el = vm.el //进行依赖收集
return el + 222
```

这样它就能解决下面的三元表达式问题。

```
a ? a: b
```

这里扼要说明一下什么叫依赖收集，就是将表达式的变量抽取出来，当某个变量发生改变，就执行求值函数，然后执行后面一连串操作，从而更新元素。

通常表达式里不需要if, else语句，也不要用注释（这个在文档里说明一下，要移除JS的注释太难了），但为了预防起见，所有干扰因素都要去除掉，最后剩下逗号与变量。

（1）干掉所有正则，这个网上有现成的正则可用。

（2）干掉所有字符串，这个也有现成的正则，但不太好用，建议实现一个方法来处理。

（3）干掉所有对象的子属性。

（4）干掉所有关键字与保留字。

（5）干掉所有操作符与数字。

（6）去掉与利用hasOwnProperty进行过滤变量与局部变量（就是之前说的，i、el问题）。

比如说：

```
vm = {
  a: 1,
  b: 2
}

ms-attr-title="aaa == null ? '111': String(aaa+bbb)"
```

得到表达式。

```
aaa == null ? '111': String(aaa+bbb)
```

去掉正则，去掉字符串。

```
aaa === null ? : String(aaa+bbb)
```

去掉关键字与操作符。

```
aaa String aaa bbb
```

去重。

```
aaa String bbb
```

最后通过vm.hasOwnProperty(el)，得到aaa、bbb。

```
https://www.zhihu.com/question/29743491/answer/49301464
```

从实际代码来看，则是这样。

第一步，用正则来干掉正则字面量。

```
var rregexp = /(^|[^/])\/(?!\/)(\[.+?]|\\.|[^/\\\r\n])+\/[gimyu
```

```
]{0,5}(?=\s*($|[\r\n,.;]))))/g
```

第二步，用正则或函数干掉字符串字面量。

```
var rstring = /("|')(\\(?:\r\n|[\w\W])|(?!\1)[^\\\r\n])*\1/
```

但这个正则并不完美，也尝试从各种语法高亮库中找其他正则，也不如意。于是后来笔者又写了这个方法，里面的dig参数就是上一节的dig函数。这种一个个字符串进行读入的方法，也就是经典的分词手段，原始并高效。angular的源码里整个parser就是这样实现的。

```
function clearString(str, dig) {
    var array = readString(str)
    for (var i = 0, n = array.length; i < n; i++) {
        str = str.replace(array[i], dig)
    }
    return str
}

function readString(str) {
    var end, s = 0
    var ret = []
    for (var i = 0, n = str.length; i < n; i++) {
        var c = str.charAt(i)
        if (!end) {
```

```
            if (c === "'") {
                end = "'"
                s = i
            } else if (c === '"') {
                end = '"'
                s = i
            }
        } else {
            if (c === '\\') {
                i += 1
                continue
            }
            if (c === end) {
                ret.push(str.slice(s, i + 1))
                end = false
            }
        }
    }
    return ret
}
```

第三步，干掉子级属性。也是正则，参观上一节的正则。

第四步是去掉关键字与保留字，这个很简单，我们先弄一个hashmap，然后用正则进行抠空处理。

```
//avalon.oneObject方法见第一章
var keyMap = avalon.oneObject("break,case,catch,continue,debugger,default,delete,do, else,false," +
        "finally,for,function,if,in,instanceof,new,null,return,switch,this," +
```

```
        "throw,true,try,typeof,var,void,while,with," + /* 关键字
*/
        "abstract,boolean,byte,char,class,const,double,enum,exp
ort,extends," +
        "final,float,goto,implements,import,int,interface,long,
native," +
        "package,private,protected,public,short,static,super,sy
nchronized," +
        "throws,transient,volatile")
//-----------
"aaa,+,var, undefined,eee,,".replace(/\w+/g, function(key){
    if(keyMap[key]){//这就是抠空得理
        return ' '
    }
    return key
})
```

第五步干掉操作符与数字，也是正则。

```
var rnumber = /\b\d[^,]*/g
//from prism.js
var rnumber = /\b-?(?:0x[\da-f]+|\d*\.?\d+(?:e[+-]?\d+)?)\b/ig
//from jQuery.js
var rnumber = /[+-]?(?:\d*\.|)\d+(?:[eE][+-]?\d+|)/g
var roperator = /--?|\+\+?|!=?=?|<=?|>=?|==?=?|&&?|\|\|?|\?|\*|
\/|~|\^|%/g
//大中小括号、冒号、逗号、点号
var rpunctuation =  /[{}[\];(),.:]/g
```

因此整个过程就是与正则“斗智斗勇”的过程。当然也可以不用正则，但不用正则，使用函数方式实现parser，代码量太多了，angular与ractive就是被

parser拖累。并且随着EsmaScript一年升一次级的步伐，每年都加入新的语法，parser是越来越复杂，你总有遗漏的风险。因此不建议实现一个完整的parser。只是对自己感兴趣的变量进行抽取前置就行了。

随着技术的发展，变量抽取也正被另一种技术所替换——路径抽取 。

变量抽取是为动态依赖收集服务。

路径抽取是为静态依赖收集服务。

动态依赖收集，要对求值函数进行深改造。要执行一下求值函数，让里面前端的赋值语句被执行，从而触发访问器属性的getter方法。因此它有一个硬性要求 ，vm必须是一个特殊的对象。这个对象的所有属性或大部分属性要转换为访问器属性，或者像knockout将属性变成一个方法。像angular那

样，如果vm是一个普通对象，不存在访问器属性，里面前置赋值语句是没有意义的。因此它就发展出另一种技术，路由抽取，对应的，它能监听子级对象的属性变动。

```
vm.$watch(path, callback)
```

比如说，一个表达式是这样的。

```
aaa.bbb+aaa.ccc
```

那么它里面会产生两个$watch回调。

```
vm.$watch("aaa.bbb", callback)
vm.$watch("aaa.ccc", callback)
```

angular如何知道aaa.bbb、aaa.ccc发生了变化呢？这简单，它将aaa对象复制了一份，然后新旧aaa对象进行diff，发现不一致，就触发对应路径的

回调。这个步骤，angular称之为脏检测。效率无法与经典的MVVM精准属性监控相比。angular也尝试过精准属性监控，为此还放弃了浏览器兼容性，特意让干爹Chrome搞了一个Object.observe的新API。但其他浏览器不跟Chrome玩，最后也不了了之。

抽取变量与抽取路径步骤基本一致。有兴趣可以看一下avalon1.5.8的实现。avalon1.5是动态依赖收集与静态依赖收集都用上。

在ractivejs或polymer等库，路径抽取还存在通配符的情况。

```
vm.$watch("array.*.aaa", callback)
```

这个就更加复杂了。因此在生成求值函数时，动态模板比静态模板复杂多。这一切都是为了最小化刷新。

15.2.2　刷新函数

刷新函数是内置的视图刷新方法。有多少种指令，就有多少种方法。换言之，这一节主要介绍通用的指令类型。

指令通常分为两大类型，逻辑指令与普通指令。

逻辑指令的操作对象是元素节点，比如说ms-if、ms-for（avalon2系）、ng-if、ng-repeat、ng-switch（angular系）、each、if、ifnot、with（knockout系）。

普通指令则是操作元素的某个属性（ms-attr, ng-attr）、某个样式（ms-css, ng-css）、nodeValue（ms-text, ng-text, 双花括号）、innerHTML（ms-html, ng-html）、某种事件（ms-on-click, ng-on-click，它们大多数可以缩写成ms-

click, ng-click）。在ng2时代起，所有MVVM框架也不约而同地使用冒号代替各种前端，于是ms-attr、ng-attr变成：attr。

只要我们知道了绑定对象的type属性，就一下子能找到对象的刷新函数。简单的刷新函数，如text指令的可以写成这样。

```
function text(node, value, binding){
   node.replaceData(0, value.length, value)
   //node为文本节点， replaceData可以参看这里
   //http://www.jb51.net/w3school/xmldom/dom_text.htm
}
```

或者写成这样。

```
function text (node, value, binding){
   node.nodeValue = value
}
```

html指令则是这样。

```
function text (node, value, binding){
   node.innerHTML = value
```

```
}
```

或者使用jQuery。

```
function text (node, value, binding){
   $(node).html(value)
}
```

像attr、css指令则更是如此，它们内部存在复杂兼容性处理，要么你使用jQuery，要么自己实现一套attr、css方法。因此angular1内部存在一个jqLite的轻量库。这时你可以把之前章节（属性模块，样式模块，事件模块）学到的东西搬过来。

这些还是比较简单的指令，刷新函数还是很好实现，像if、for指令就麻烦了。

if指令要求表达式的值为真的时，渲染原元素；为假时，将原元素移出DOM树，原位置用一个注释节点占位；再次为真的，将原元素放回去

（avalon, knockout）或是重新生成一个元素放回去（angular）。

for指令则会生成更多占位用的注释节点，有的框架称之为路标系统，有的框架称之为锚点系统。

angular的ng-repeat指令，注意每一项中间也存在注释节点隔开。

```
 <li class="animate-repeat" ng-repeat="friend in friends | filter:q as results">
      [{{$index + 1}}] {{friend.name}} who is {{friend.age}} years old.
</li>
```

```
<!-- ngRepeat: friend in friends | filter:q as results -->
<li class="animate-repeat ng-binding ng-scope" ng-repeat="friend in frie
results">
      [1] John who is 25 years old.
    </li>
<!-- end ngRepeat: friend in friends | filter:q as results -->
<li class="animate-repeat ng-binding ng-scope" ng-repeat="friend in frie
results">
      [2] Johanna who is 28 years old.
    </li>
<!-- end ngRepeat: friend in friends | filter:q as results -->
<li class="animate-repeat ng-binding ng-scope" ng-repeat="friend in frie
results">
      [3] Mary who is 28 years old.
    </li>
<!-- end ngRepeat: friend in friends | filter:q as results -->
<li class="animate-repeat ng-binding ng-scope" ng-repeat="friend in frie
results">
      [4] Erika who is 27 years old.
    </li>
<!-- end ngRepeat: friend in friends | filter:q as results -->
```

avalon2的ms-repeat指令。

```
<ol>
    <li ms-for='($key, $val) in @object'>{{$key}}::{{$val}}</li
>
</ol>
```

```
▼ <ol> == $0
    <!--ms-for:($key, $val) in @object-->
    <li>a::11</li>
    <!--for054689433025-->
    <li>b::22</li>
    <!--for054689433025-->
    <li>c::33</li>
    <!--for054689433025-->
    <li>d::44</li>
    <!--for054689433025-->
    <li>e::55</li>
    <!--for054689433025-->
    <!--ms-for-end:-->
  </ol>
```

有的MVVM框架，不使用注释节点作为锚点，

改成空白节点。这就要希望浏览器的兼容性较好，因为中间分隔用的锚记需要知道它是从属某个for循环里面，肯定要通过nodeValue或其他什么来区分。比如angular，它们都是`end` 加上用户的指令属性值，avalon则是使用同一个UUID（`for\d+` ）。否则出现双重循环或多重循环，就会乱套。空白节点只是一个假象，让那些喜欢在控制台观察的人，觉得这个框架不会产生这么多乱七八糟的注释节点，肯定性能更优了。但生成一个空白节点与生成一个注释节点，对浏览器而言没什么区别，都需要这么多重原型继承。空白节点，你还得手动在上面添加一个特殊的属性，属性值为ms-for或ng-for的属性值，来区分循环。

```
var anchor = document.createTextNode('')
anchor.xxx = '($key, $val) in @object'
```

在IE6～IE8下，文本节点与注释节点是不能添

加自定义属性，这就是avalon不采用此方案的缘故。

其实使用注释节点有一个好处，我们可以直接使用原生方法得到所有锚点。

```
    var queryComments = DOC.createTreeWalker ? function(parent)
 {
        var tw = DOC.createTreeWalker(parent, NodeFilter.SHOW_C
OMMENT, null, null),
                comment, ret = []
        while (comment = tw.nextNode()) {
            ret.push(comment)
        }
        return ret
    } : function(parent) {
        return parent.getElementsByTagName("!")
    }
//或者
    var getComments = function(parent, array) {
        var nodes = parent.childNodes
        for (var i = 0, el; el = nodes[i++]; ) {
            if (el.nodeType === 8) {
                array.push(el)
            } else if (el.nodeType === 1) {
                getComments(el, array)
            }
        }
    }
    var queryComments = function(parent) {
        var ret = []
        getComments(parent, ret)
        return ret
    }
```

在ember.js中，则干脆使用script节点做锚点。

有了锚点系统，当你的数组由a、b、c变成c、b、a时（这时通常会介入一个非常复杂的diff算法，如最短编辑距算算法，计算出最少移动的步骤，具体可以看一下knockout源码。像react，则强制要求你给一个不重复的key，这时通过hash得到移动位置。像avalon1、avalon2它们的实现也是各显神通的），这些由锚点分开的元素就会抽出DOM树，变成一个文档碎片，插入到目标位置的锚点的前面。

有关for指令的实现可以参看这两个链接。

```
https://github.com/RubyLouvre/avalon/issues/372
https://github.com/RubyLouvre/avalon/blob/2.1.6/src/directives/
for.js
```

接着下来说事件指令与duplex指令。

事件指令有点特别，它要求求值函数总是返回同一个函数，当然你也可以是不同的函数实例，但要你标识它与之前的是否一致。

```
function old(){
    return aa + 1
}
function neo(){
    return aa + 1
}
```

由于它们的结构一致，因此你用==号来比较是没有意义的，但你又不能再绑一次。这时只能比较表达式。如果生成它们的表达式一致，那么只需绑一次。这时另一个问题来了，怎么知道这个函数已经被绑上呢？于是我们还是将前几章的基于UUID的事件系统搬上来。此事件系统会访问函数上面有没有uuid属性，没有就会加一个。因此我们可以抢先把binding.expr放到上面去，这样就实现绑定一次事件的效果。

而duplex指令，它也是这样，不同的是，它可以绑定多个事件，尤其要兼容IE6～IE8的情况下。此外，duplex是双工指令，顾名思义，它是会反过回操作VM，因此它不但有get方法，还有set方法。在avalon中，它实现对元素的value属性劫持，当元素的value不是在事件回调中被修改的情况下，也能通知框架更新VM。

图15-6和图15-7是angular与avalon的指令设计图。

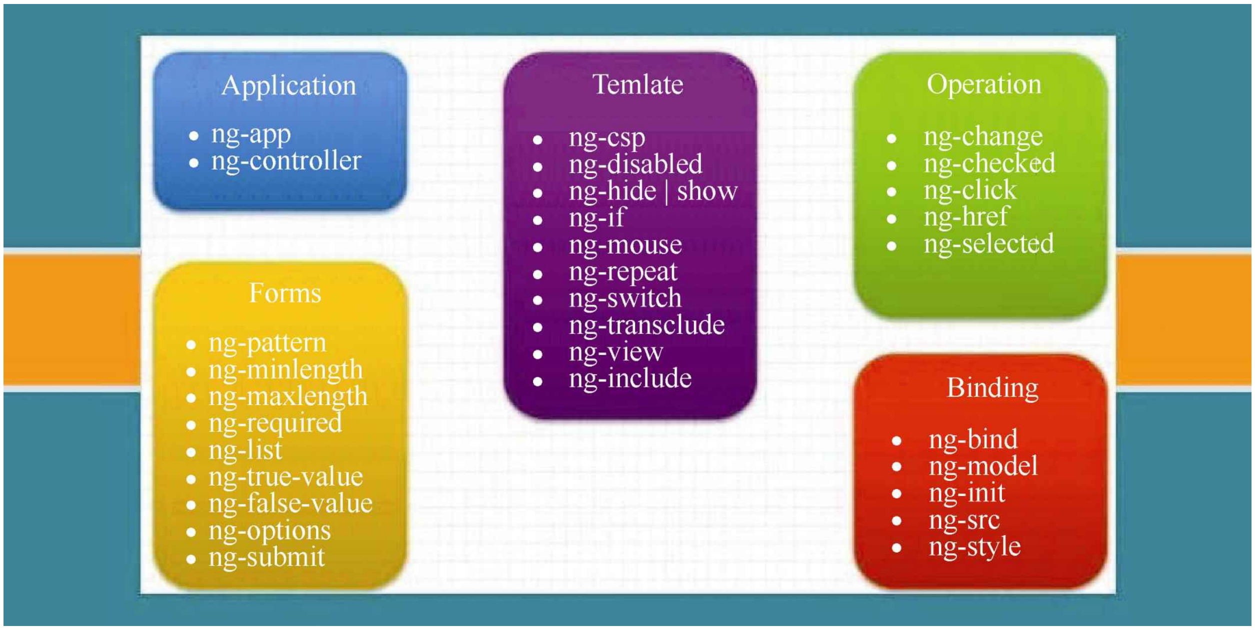

Application
ng-app
ng-controller
Forms
ng-pattern
ng-minlength
ng-maxlength
ng-required
ng-list
ng-true-value
ng-false-value
ng-options
ng-submit
Temlate
ng-csp
ng-disabled
ng-hide | show
ng-if
ng-mouse
ng-repeat
ng-switch
ng-transclude
ng-view
ng-include
Operation
ng-change
ng-checked
ng-click
ng-href
ng-selected
Binding
ng-bind
ng-model
ng-init
ng-src
ng-style

图15-6

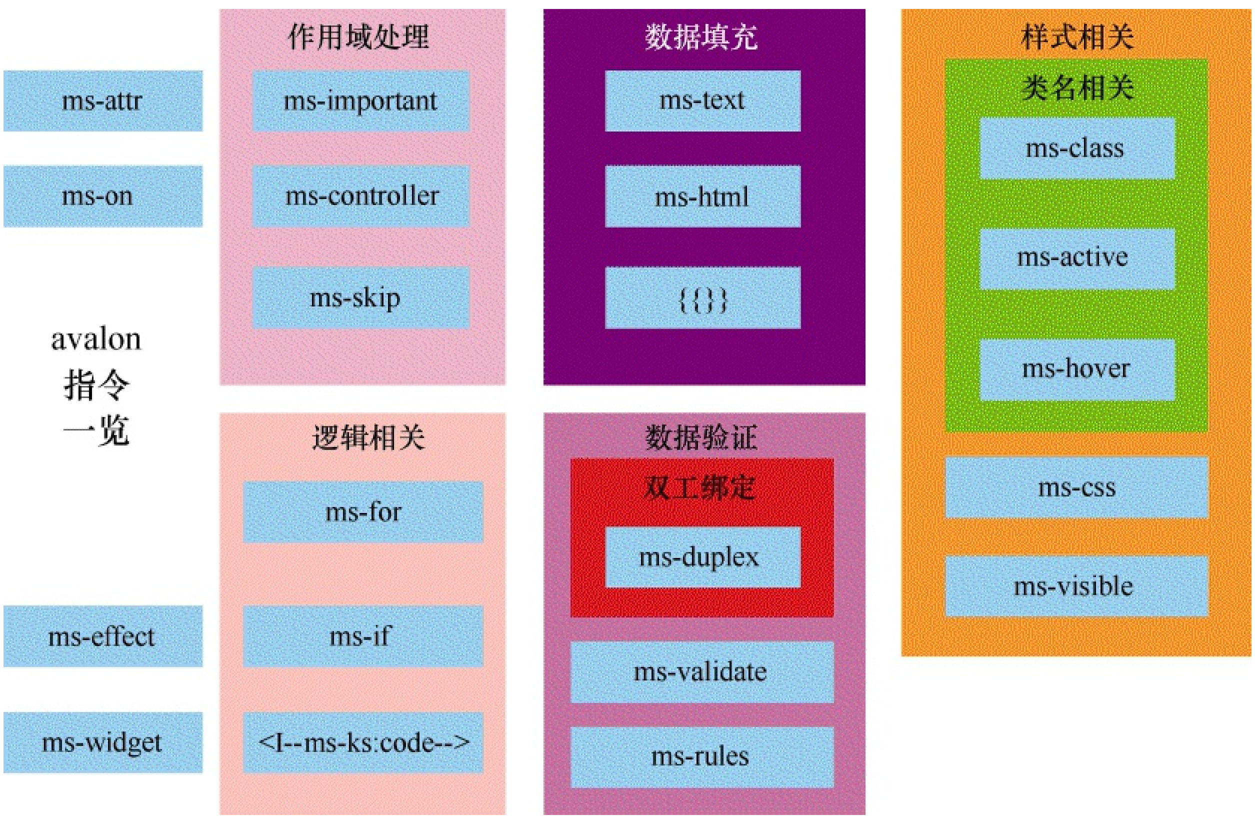

avalon
指令
一览
ms-attr
ms-on
ms-effect
ms-widget
作用域处理
ms-important
ms-controller
ms-skip
逻辑相关
ms-for
ms-if
<I--ms-ks:code-->
数据填充
ms-text
ms-html
{{}}
数据验证
双工绑定
ms-duplex
ms-validate
ms-rules
样式相关
类名相关
ms-class
ms-active
ms-hover
ms-css
ms-visible

图15-7

15.3 ViewModel

MVVM的核心在于ViewModel，根据其是否主动进行监控，又分为两大派系。avalon这样通过挖掘语言特性打造的重量对象VM，或是像angular那样依靠外部的diff机制检测自身变化的轻量对象VM。

从对象继承的原型链长度及占用内存来划分，JavaScript分为4种对象。

```
超轻量 Object.create(nulll) 没有原型
轻量  只有一重原型的普通对象 {}
重量  只有一到二重原型，或带有访问器属性的对象，avalon或vue的VM对象
超重量 存在5级原型的对象，各种DOM节点或window对象
```

但作为avalon式的VM必须用访问器属性构成。

在第4章《类工厂》中，我们介绍了如何使用

Object.defineProperty为一个对象添加一个访问器属性，其实在更早期的时候，Firefox就添加了__lookupSetter__、__lookupGetter__、__defineSetter__、__defineGetter__ 4个方法。前2个合成现在的Object.getOwnPropertyDescriptory方法，后4个合成现在的Object.defineProperty方法。这也成了旧的W3C浏览器实现avalon式VM的解决之道。

在旧的IE中怎么办？虽然Object.defineProperty是在IE8提出来的，但它不能应用于普通对象，只能用于元素节点上。基于前人做了许多探索，有如下几种方法。

（1）用元素节点代替普通对象，dom.onpropertychange 事件可以侦听setter，而且没有getter，没有getter就无法进行动态依赖收集。不过现在发展出静态依赖收集，这也没什么了，但最

坑的是我们需要对用户对象的属性名进行限制，比如不能等于元素的固有属性名。由于元素节点有七重原型链，每重都占用大量属性名与方法名，中奖概率比六合彩高多了，因此此方案早早出局。

（2）用预编译手段，将项目的源码中obj[member] = val 替换成比如superSetter（obj, member, val），var a = obj[member] 替换成var a = superGetter(obj, member) 。但目前没人尝试这么干，太累人了。

（3）使用VBScript，它拥有3种类似的语句Set、Get、Let，可以实现相同的功能，坑主要有3个。1．VBScript对象一旦生成，就不能再添加新成员，否则就抛错。由于VM不单使用方法器属性，需要做其他大量处理，添加与删除一个成本太高。基本上都要求先定义后使用，这个缺憾可以承受。2．属性名不区分大小写，这个需要注意一下就

行。3．会占用少量的属性名，目前发现这3个：err，type，me 。4．函数的this不再指向对象，这个可以使用bind方法搞定。

下面是dojo实现的版本。

```
http://download.dojotoolkit.org/release-1.9.7/dojo-release-1.9.7/dojox/lang/observable.js.uncompressed.js
```

而 Avalon 实现的版本是目前最实用的，大量应用于生产环境。

（4）使用flash，flash的语言特征一直是JavaScript抄袭对象，早些年的ECMA4就是忠诚拥趸。尽管falsh在PC上基本都安装了，但不怕一万，最怕万一。目前也没有看到有人使用此方案。

Object类位于动作脚本类层次结构的根处。此类包含JavaScript Object类所提供功能的一小部分，如图15-8所示。

Object 类位于动作脚本类层次结构的根处。此类包含 JavaScript Object 类所提供功能的一小部分。

Object 类的方法摘要

方法	说明
Object.addProperty()	在对象上创建 getter/setter 属性。
Object.registerClass()	将影片剪辑元件与动作脚本对象类相关联。
Object.toString()	将指定对象转换为字符串然后返回它。
Object.unwatch()	删除 Object.watch() 创建的监视点。
Object.valueOf()	返回对象的原始值。
Object.watch()	注册当动作脚本对象的指定属性更改时要调用的事件处理函数。

图15-8

说完这些悲惨的兼容方案，我们看一下有没有新的API可以用吧。Object.defineProperty只对取值赋值操作敏感，如果我们为元素添加一个新属性呢，删除一个属性呢，它就无能为力。于是又是Firefox为我们带来一些好玩的魔术。

15.3.1　Proxy

这是ES6的新东西，完全不是同一个次元的动物，觉得是来自ruby星球上的。之前，谷歌还捎带了一个“私货”叫Object.observe，一大堆人来捧臭脚。但其他浏览器不理帐，只活了两年，从

Chrome36～Chrome49。说回Proxy，这是一个逆天的东西，不像Object.defineProperty，只是对=号敏感。什么delete、for in、in、new，还是方法调用都能感应到！这是一个非常顶级的自省机制，要求浏览器对普通对象内部添加大量的钩子。JS之父所在的Firefox很早就搞出这东西，那些还是规范1，后来标准化后，接口有点不一样。Chrome也是规范1，规范2都实现过，不过中途断线了几个版本。

Proxy是一个构造函数，使用new Proxy创建代理器，第一个参数为一个对象，第二个参数也为一个对象，返回被包裹后的代理器，我们使用基本的get和set写一个demo。

```
var obj = new Proxy({}, {
    get : function( target , prop ) {
        console.log(prop, '进行取值');
        return target[prop];
    },
    set : function( target, prop, value) {
        console.log(prop, value, target[prop], '打印新旧值');
        target[prop] = value;
    }
});
obj.aaa = 1;
```

```
obj.aaa;
obj.bbb = 2
obj.bbb = 3
```



如果要监听属性被删除，可以在第二个对象添加一个deleteProperty方法。它会将原对象与要删除的属性传给你，实际上它什么也没有做，你需要自己进行删除。

```
var obj = new Proxy({}, {
        get: function (target, prop) {
            console.log(prop, '进行取值', target[prop]);
            return target[prop];
        },
        deleteProperty: function (target, prop) {
            console.log(prop, '被删除了')
            delete target[prop]
            return true //如果决定删除则返回true
        },
        set: function (target, prop, value) {
            console.log(prop, value, target[prop], '打印新旧值');
            target[prop] = value;
        }
    });
    obj.aaa = 1;
    obj.aaa;
    delete obj.aaa
    obj.aaa
```

此外还有其他接口，但基本上与VM无关了。作为介绍，也一并列出来吧。

```
    handler.getPrototypeOf()
    handler.setPrototypeOf()
    handler.isExtensible()
    handler.preventExtensions()
    handler.getOwnPropertyDescriptor()
    handler.defineProperty()
    handler.has()
    handler.get()
    handler.set()
    handler.deleteProperty()
    handler.ownKeys()
    handler.apply()
    handler.construct()
```

大家可以访问以下网页了解其更多用法。

```
http://www.cnblogs.com/diligenceday/p/5467634.html
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Referen
ce/Global_Objects/Proxy
```

15.3.2　Reflect

在angular2的内部就是使用Reflect实现VM。它的行为与Proxy很相似，可以说Proxy的第二个参数

对象有什么方法，它就有什么方法。

Reflect对象的设计目的有如下4个。

（1）将Object 对象的一些明显属于语言内部的方法（比如Object.defineProperty）放到Reflect对象上。现阶段，某些方法同时在Object和Reflect对象上部署，未来的新方法将只部署在Reflect对象上。

（2）修改某些Object 方法的返回结果，让其变得更合理。比如，Object.defineProperty(obj, name, desc)在无法定义属性时，会抛出一个错误，而Reflect.defineProperty(obj, name, desc)则会返回false。

```
旧写法
try {
  Object.defineProperty(target, property, attributes);
  //uccess
} catch (e) {
  //failure
}
```

```
//新写法
if (Reflect.defineProperty(target, property, attributes)){
  // success
} else {
  // failure
}
```

（3）让Object操作都变成函数行为。某些Object操作是命令式，比如name in obj和delete obj[name]，而Reflect.has(obj, name)和Reflect.deleteProperty(obj, name)让它们变成了函数行为。

```
//方法调用
Math.floor(1.75)
Reflect.apply(Math.floor, undefined, [1.75]);
//添加操作
var obj = {};
obj.x = 7
Reflect.defineProperty(obj, "y", {value: 7});
//删除属性
var obj = { x: 1, y: 2 };
delete obj.x //true
Reflect.deleteProperty(obj, "7"); // true
obj; // {}
//读取属性
var obj = { x: 1, y: 2 };
obj.x // 1
Reflect.get(obj, "x"); // 1
//实例化
var obj = new Foo(1, 2);
var obj = Reflect.construct(Foo, [1, 2]);
var d = new Date(1776, 6, 4)
var d = Reflect.construct(Date, [1776, 6, 4]);
```

```
//获取其所有键名
Object.getOwnPropertyNames({z: 3, y: 2, x: 1})
Reflect.ownKeys({z: 3, y: 2, x: 1}); // [ "z", "y", "x" ]
```

（4）Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。这就让Proxy对象可以方便地调用对应的Reflect方法，完成默认行为，作为修改行为的基础。也就是说，不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

Reflect对象的方法清单如下，共14个。

```
Reflect.apply(target,thisArg,args)
Reflect.construct(target,args)
Reflect.get(target,name,receiver)
Reflect.set(target,name,value,receiver)
Reflect.defineProperty(target,name,desc)
Reflect.deleteProperty(target,name)
Reflect.has(target,name)
Reflect.ownKeys(target)
Reflect.enumerate(target)
Reflect.isExtensible(target)
Reflect.preventExtensions(target)
Reflect.getOwnPropertyDescriptor(target, name)
Reflect.getPrototypeOf(target)
Reflect.setPrototypeOf(target, prototype)
```

更多用法可以看这里。

```
http://www.cnblogs.com/diligenceday/p/5474126.html
```

15.3.3　avalon的ViewModel设计

不是有了Object.defineProperty在Proxy或Reflect中，放进一个对象就new出一个ViewModel出来。只能说，它们是必要条件。我们需要将要监听的属性变成访问器属性，所有访问器属性都是共用同一套setter、getter方法。getter里面做依赖收集（不是必须的），setter里做视图刷新 或触发该属性的$watch回调。在此之前，我们需要完成一套观察者模式，就是github中常见的EventEmitter库。

但这些库的订阅数组都是放函数。如果我们要放绑定对象，需要改造一下，并且改成`$watch`, `$fire` 接口。

```
var EventBus = {
```

```
    $watch: function (type, callback) {
        var binding = callback
        if (typeof callback === "function") {
            binding = {
                expr: type,
                update: callback
            }
        }
        var bus = this.$events
        var list = bus[type]
        if (!list) {
            list = bus[type] = []
        }
        function unwatch() {
            avalon.Array.remove(list, binding)
            if(!list.length){
               delete bus[type]
            }
        }
        list.push(binding)
        return unwatch
    },
    $fire: function (type, value) {
        var list = this.$events[type]
        if (list && lsit.length) {
            for (var i = 0, obj; obj = list[i++];) {
                obj.update()
            }
        }
    }
}
```

然后我们再为它添加一个$id，用于标记这个VM是作用于页面某个元素上的。

```
var vm = avalon.define({
    $id: "test",
    aaa: 1,
```

```
    bbb: 2
})

<div ms-controller="test">{{@aaa}}</div>
```

我们看avalon.define的一个简单实现。

```
avalon.define = function(obj) {
    var vm = {}
    var other = {}
    for (var name in obj) {
        if (typeof obj[name] !== 'function' && name.charAt(0) !
== '$') {
            (function (key, value) {
                function get(){
//在avalon1.4,1.5中这里会进行动态依赖收集，详见这里
//https://github.com/RubyLouvre/avalon/blob/1.5/src/10%20depend
ency.js
                    return get._value
                }
                get._value = value
                Object.defineProperty(obj, key, {
                    set: function (newValue) {
                        if(newValue !== get._value){
                            get._value = newValue
                            if(vm.$hashcode)
                               vm.$fire(key, newValue)
                        }
                        return newValue
                    },
                    get: get
                })
            })(name, obj[name])
        }else{
            other[name] = obj[name]
        }
    }
    for(var name in other){
        vm[name] = other
    }
```

```
    vm.$events = {}
    vm.$hashcode = new Date  - Math.random()
    vm.$fire = EventBus.$fire
    vm.$watch = EventBus.$watch

    return avalon.vmodels[vm.$id] = vm
}
```

此外，你可以添加更多以$ 开头的属性方法，来增强它的功能。在avalon、angular等库中，$ 开头的属性方法都是框架自用的。avalon2的一个简单的VM是藏了许多不可遍历的$xxx 属性方法。



那么如何将绑定属性放进vm.$events.aaa数组中呢？这就要靠扫描机制，从上到下扫描。

```
avalon.scan = function (el, vm) {
    scanNodes([el], vm)
}
function scanNodes(array, vm) {
    for (var i = 0, el; el = array[i++]; ) {
        switch (el.nodeType) {
            case 1:
                scanTag(el, vm)
                break
            case 3:
                scanText(el, vm)
                break
```

```
        }
    }
}

function scanTag(el, vm){
    var id = el.getAttribute('ms-controller')
    if(id && avalon.vmodels[id]){
        var vm2 = avalon.vmodels[id]
        if(vm && vm2 && vm == vm2){
           vm = mergeVM(vm,vm2)
        }else{
           vm = vm2
        }
        el.removeAttribute('ms-controller')
    }
    var bindings = scanAttrs(el,vm)
    for(var i = 0, b; b = bindings[i++];){
        vm.$watch(b.expr, b) //重点
    }
    if(el.children && el.children.length){
        scanNodes(el.children, vm)
    }
}
function scanText(){
    // 用正则检测是否有花括号
    // 有则转换为绑定对象
    // 并进行vm.$watch
}
function scanAttrs(){
    //遍历el.attributes中所有对象,看name是否以ms-开头
}
```

图15-9里面用到一个mergeVM方法，其实很简单，就是将两个VM合并成一个新的VM。使用Object.getOwnPropertyDescriptor或者更新的Object.getOwnPropertyDescriptors，就能得到所有访

问器属性的定义对象，然后合成。如果是古老浏览器，我们可以将访问器属性放到一个叫$accessors对象上。

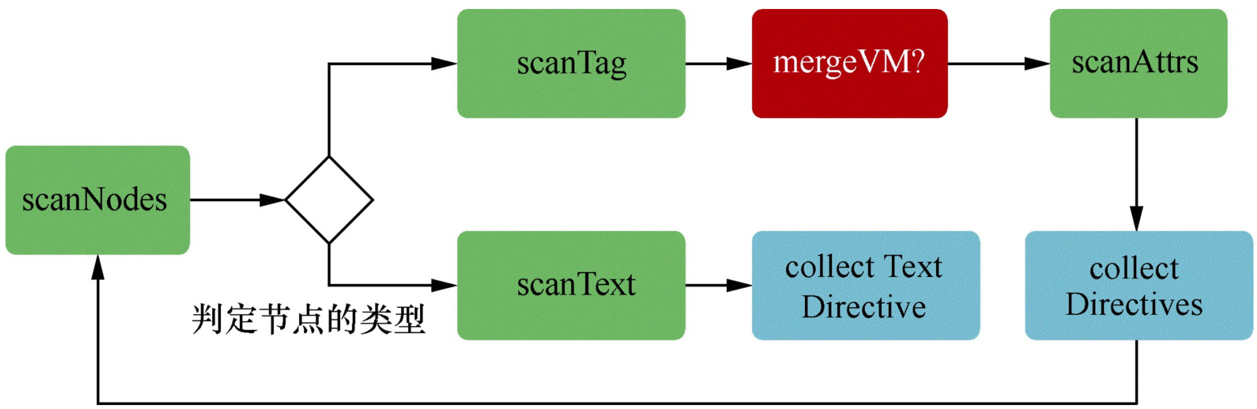


图15-9

现在我们这个VM是很简单的，它只支持一重属性。如果属性的属性也是对象呢？这个我们需要将这define方法递归一下不就行了吗！对于数组的监控，业界流行的方法是重写数组的大部分方法，然后再加上一些移除数组的方法。

至此，avalon内部各种概念的关系图，如图15-10所示。

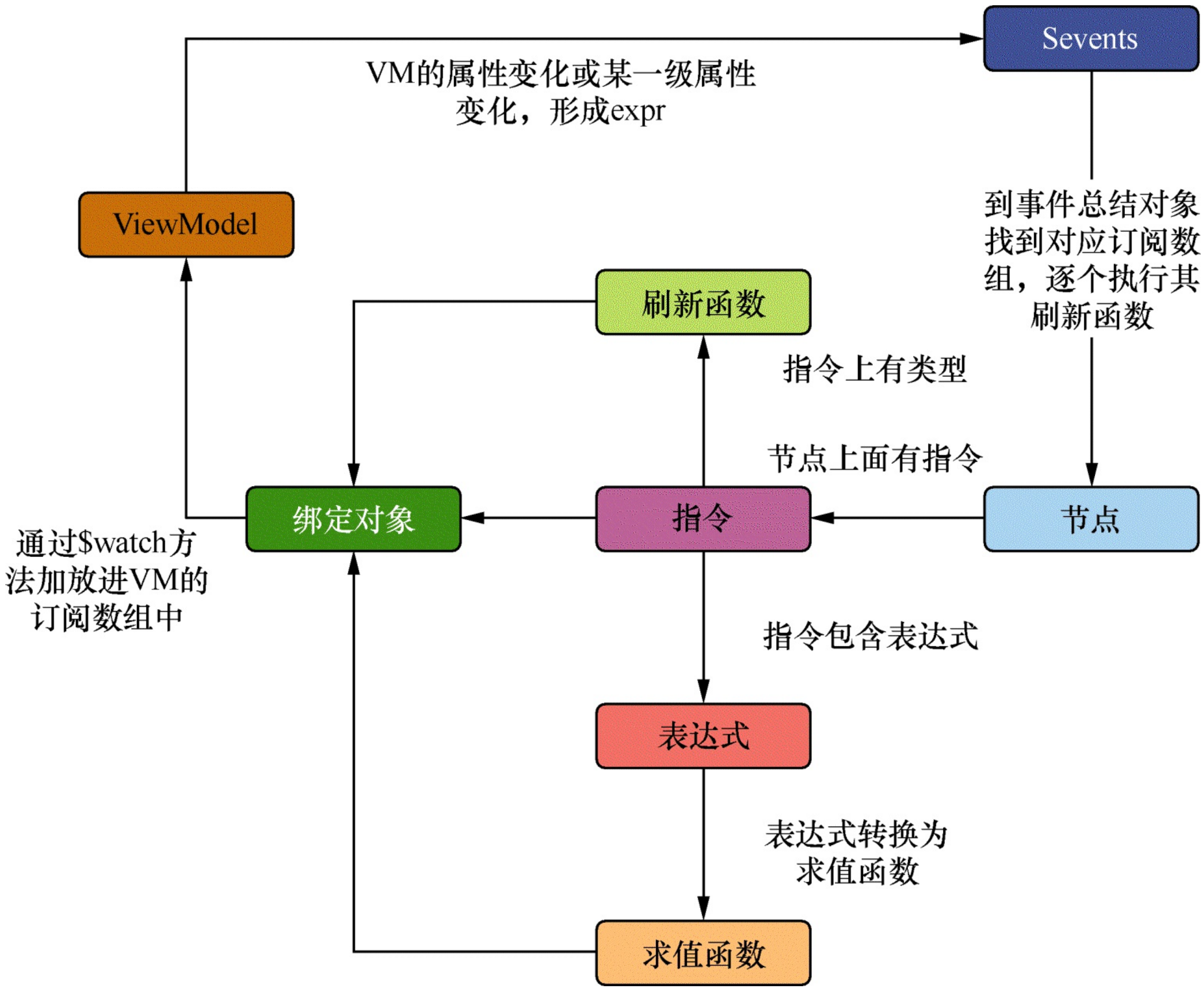


图15-10

15.3.4　angular的ViewModel设计

angular的ViewModel有一个专门的官方术语叫`$scope`，它只是一个普通构造器（Scope）的实例。换言之，它是一个普通的JS对象。为了实现

MVVM框架通常宣传的那种“改变数据即改变视图”的魔幻效果，它得装备上更多更强大的外挂。

```
<div ng-app="myApp" ng-controller="myCtrl">

名: <input type="text" ng-model="firstName"><br>
姓: <input type="text" ng-model="lastName"><br>
<br>
姓名: {{firstName + " " + lastName}}

</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName = "John";
    $scope.lastName = "Doe";
});
</script>
```

app.controller会产生一个`$scope`对象，这个`$scope`是传进去的。

```
var $scope = new Scope();
$scope.firstName = 'Jane';
$scope.lastName = 'Smith';
```

相对于avalon将所有VM扁平化地放到

avalon.vmodels中，angular则倾向将$scope 对象以树的形式组织起来。

```
function Scope() {
this.$id = nextUid();
      this.$$phase = this.$parent = this.$$watchers =
                     this.$$nextSibling = this.$$prevSibling =
                     this.$$childHead = this.$$childTail = null
;
      this.$root = this;
      this.$$destroyed = false;
      this.$$listeners = {};
      this.$$listenerCount = {};
      this.$$watchersCount = 0;
      this.$$isolateBindings = null;
}
```

其中$parent 、$$nextSibling 、$$prevSibling 、$$childHead 、$$childTail 、$root 是指向其他$scope 对象。$$watchers 是绑定对象的订阅数组，$$watchersCount 是其长度，$$listeners 是放手动触发的函数，$$listenerCount 是其长度。

由于angular是一个普通的JS对象，当属性发生

变化时，它本身不可能像avalon那么灵敏地跑去$fire 。于是它实现了一套复杂的$fire 方法，但它不叫$fire ，叫做$digest 。

换言之，avalon的$watch 对应angular的$watch ，此外，它还有$watchGroup , $watch Collection 。avalon的$fire方法对应angular的$digest 。为了安全，它外面还有$applyAsync 、$apply 、$evalAsync 等几个壳函数。它们共同构成angular的监控系统。$watch 和$digest 是相辅相成的，两者一起，构成了angular作用域的核心功能，数据变化的响应。

先看$watch 方法，传参比avalon复杂多，但结果都是返回一个移除监听的函数。

```
Scope.prototype.$watch: function(watchExp, listener, objectEquality, prettyPrintExpression) {
  //将表达式转换为求值函数
    var get = $parse(watchExp);

    if (get.$$watchDelegate) {
```

```
      return get.$$watchDelegate(this, listener, objectEquality
, get, watchExp);
    }

    var scope = this,
    //所有绑定对象都放在一个数组中，因此存在性能问题
        array = scope.$$watchers,
    //构建绑定对象
        watcher = {
          fn: listener,//刷新函数
          last: initWatchVal,//旧值
          get: get,//求值函数
          exp: prettyPrintExpression || watchExp,//表达式
          eq: !!objectEquality// 比较方法
        };

    lastDirtyWatch = null;

    if (!isFunction(listener)) {
      watcher.fn = noop;
    }

    if (!array) {
      array = scope.$$watchers = [];
    }
    array.unshift(watcher);
    incrementWatchersCount(this, 1);

    return function deregisterWatch() {//移除绑定对象
      if (arrayRemove(array, watcher) >= 0) {
        incrementWatchersCount(scope, -1);
      }
      lastDirtyWatch = null;
    };
},
```

而$digest则复杂多了，我们先实现一个它的简化版，遍历其所有绑定对象，执行其刷新函数。

```
Scope.prototype.$digest = function() {
  var list = this.$$watchers || []
  list.forEach(function(watch) {
    var newValue = watch.get()
    var oldValue = watch.last;
    if (newValue !== oldValue) {
      watch.fn(newValue, oldValue, self);
    }
    watch.last = newValue;
  })
}
```

到目前为止，它的逻辑与 avalon的一样，但要明白一点，avalon的监控是智能的，如果更新A属性，导致了B属性也发生变化，那么avalon也连忙更新B涉及的视图。而angular的`$$watcher` 里面都是一个个普通对象，假如里面有A、B两个对象。先执行A，A值没有变化，再执行B，B变化了，但B在变化的同时，也修改了A值。但这时，循环已经完毕。B涉及的视图变动，A没有变动，这就不合理了。因此，我们需要在某个绑定对象发生了一次改动后，再重新检测这个数组。

我们把现在的`$digest` 函数改名

为$$digestOnce，它把所有的监听器运行一次，返回一个布尔值，表示是否变更了。

```
Scope.prototype.$$digestOnce = function() {
  var self  = this;
  var dirty;
  _.forEach(this.$$watchers, function(watch) {
    var newValue = watch.get();
    var oldValue = watch.last;
    if (newValue !== oldValue) {
      watch.fn(newValue, oldValue, self);
      dirty = true;
    }
    watch.last = newValue;
  });
  return dirty;
};
```

然后，我们重新定义$digest，它作为一个“外层循环”来运行，当有变更发生的时候，调用$$digestOnce。

```
Scope.prototype.$digest = function() {
  var dirty;
  do {
    dirty = this.$$digestOnce();
  } while (dirty);
};
```

`$digest` 现在至少每个监听器运行一次了。如果第一次运行完，有监控值发生变更了，标记为dirty，所有监听器再运行第二次。这会一直运行，直到所有监控的值都不再变化，整个局面稳定下来了。

但这里面有一个风险，比如A的求值函数里会修改B，B的求值函数又修改A，那么大家都无法稳定下来，不断死循环。因此我们得把digest的运行控制在一个可接受的迭代数量内。如果这么多次之后，作用域还在变更，就勇敢放手，宣布它永远不会稳定。在这个点上，我们会抛出一个异常，因为不管作用域的状态变成怎样，它都不太可能是用户想要的结果。

迭代的最大值称为TTL（short for Time To Live），这个值默认是10，可能有点小（我们刚运行了这个digest 成千上万次），但是记住这是一个

性能敏感的地方，因为digest经常被执行，而且每个digest运行了所有的监听器。

```
Scope.prototype.$digest = function() {
  var ttl = 10;
  var dirty;
  do {
    dirty = this.$$digestOnce();
    if (dirty && !(ttl--)) {
      throw "10 digest iterations reached";
    }
  } while (dirty);
};
```

但这只是模拟了angular的`$digest` 的冰山一角，可见没有访问器属性这高阶魔法，想实现MVVM是非常麻烦与复杂，并且用户使用起来也别扭。

有关`$digest` 的源码与解决可见这里。

```
https://github.com/angular/angular.js/blob/v1.5.8/src/ng/rootScope.js
http://www.cnblogs.com/xuezhi/p/4897831.html
```

我们再看$digest 是怎么与angular的ng-model 关联在一起。

ng-model指令有一个`$post` 方法，它在里面进行绑定事件，如果用户提供了updateOn这个选项，选项是一些事件名，那么它就为元素绑定对应的事件，否则就绑定blur方法。

```
post: function ngModelPostLink(scope, element, attr, ctrls) {
  var modelCtrl = ctrls[0];
  if (modelCtrl.$options.getOption('updateOn')) {
    element.on(modelCtrl.$options.getOption('updateOn'), functi
on(ev) {
      modelCtrl.$$debounceViewValueCommit(ev && ev.type);
    });
  }
  function setTouched() {
    modelCtrl.$setTouched();
  }
  element.on('blur', function() {
    if (modelCtrl.$touched) return;

    if ($rootScope.$$phase) {
      scope.$evalAsync(setTouched);
    } else {
      scope.$apply(setTouched);
    }
  });
}
```

我们先看blur的回调，里面$evalAsync与$apply 方法，它们里面就会调用$digest，进行脏检测，如图15-11所示。

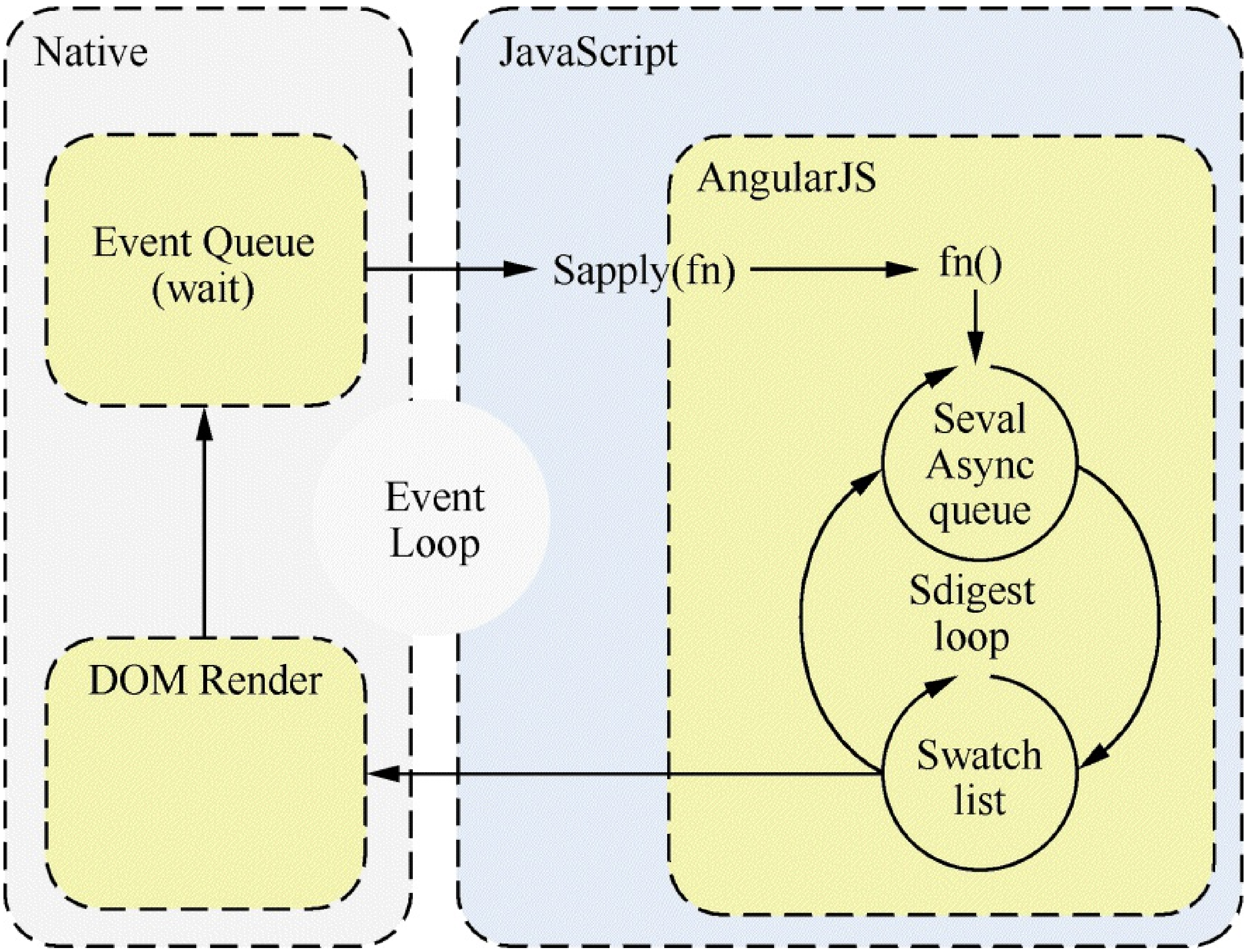


图15-11

```
$evalAsync: function(expr, locals) {
  if (!$rootScope.$$phase && !asyncQueue.length) {
    $browser.defer(function() {
      if (asyncQueue.length) {
        $rootScope.$digest();
      }
```

```
    });
  }

 //……略
},
$apply: function(expr) {
  try {
    beginPhase('$apply');
    //……略
  } finally {
    try {
      $rootScope.$digest();
    } catch (e) {
      $exceptionHandler(e);
      throw e;
    }
  }
},
```

再看$$debounceViewValueCommit 方法，里面也有一个$apply 方法。换言之，殊途同归，全部会在$digest 里面处理。

但如果许多地方同时发生改变，会不会将它搞死呢？不会，我们留意一下$digest 的源码最上方有一句beginPhase('$digest')，临结束时也有一句clearPhase()。$apply 里面也是beginPhase('$apply') 与clearPhase()，它们标

识这个$scope对象进行脏检测，直接抛错。

```
function beginPhase(phase) {
     if ($rootScope.$$phase) {
       throw $rootScopeMinErr('inprog', '{0} already in progres
s', $rootScope.$$phase);
     }

     $rootScope.$$phase = phase;
}

 function clearPhase() {
   $rootScope.$$phase = null;
 }
```

但`$apply`会将错误catch住，不让它影响程序继续运行。这就是官方推荐我们使用`$apply`驱动程序运行，而不直接用`$digest`的缘故。

通过上面的分析，avalon与angular的设计重点是不同的，avalon是忙于发掘语言特征，通过访问器中的setter与getter将其简单的观察者模式放进去。angular则忙于构建其复杂无比的观察者模式（本节没有展现其全貌，它除了**$$watchers**队列，还有**asyncQueue**队列、**postDigestQueue**队

列、**applyAsyncQueue** 队列），并且为了diff新旧值的不同，发展出一套名叫脏检测的机制。

15.4　React与虚拟DOM

React与angular几乎同时出现在人们眼帘，但React的发迹却晚多了，直到React Native发布了，才一炮而红。解决问题的痛点才是框架被采纳的关键。虽然angular也有自己的移动解决方案Ionic，但hybrid app在体积、性能等方面无法与原生app相提并论，但原生app对人员要求非常高，IOS招聘市场一直水涨船高，供不应求。React Native的Write once、run anywhere，一下子打开了局面。

在笔者眼中，开发者世界的一大悲哀是社群依据语言（甚至是生态系统）进行划分。JavaScript、Java、Objective-C、Python以及C++等，实际上，这导致了资源的巨大浪费，因为针对每个生态系统，都要开发类似的一套工具，诸如包管理器、IDE、核心函数库、知识库等。

举个具体的例子吧，在Facebook中，每个功能我们都必须实现3次：Web版、iOS版以及Android版。更糟糕的是，由于一个工程师往往难于同时掌握这些生态系统，我们通常需要3个人来实现一个功能，这真是悲哀。

为了解决该问题，笔者首先想到的是，我们需要一种单一的语言或生态系统。有了React Native，我们更趋向于 JavaScript 语言，但从宏观的角度看，哪一种语言并不重要。重要的是，只保留一种语言。“—— Facebook，Christopher Chedeau”。

回过头来，React之所以不受众，也有其原因，它做了许多大胆的创新。你可以创新一两处，但改变太多，就让人无法接受。React让人最大的垢病是JSX。

```
var Table = React.createClass({
  render: function () {
    return (
      <table><tbody>
        {this.props.data.map(function(row) {
```

```
          return (
            <tr>
              {row.map(function(cell) {
                return <td>{cell}</td>;
              })}
            </tr>);
        })}
      </tbody></table>
    );
  }
});
```

你可以说它是一种模板。但它又直接与业务逻辑混杂在一起，经过jQuery无入侵风潮洗礼的人，无法直接用这么丑陋的代码。如果它是改成这样，大家或许会好受些。

```
var Table = React.createClass({
  render: function () {
    //将模板独立出去
    return require("text!template.html")
  }
});
```

但当时webpack还不流行，前后打包方案只有browserify、grunt、glup，因此还是行不通。

facebook还是给出另一套方案，可以改成这样。

```
var Table = React.createClass({
  render: function () {
    return (
      React.DOM.table(null, React.DOM.tbody(null,
        this.props.data.map(function (row) {
          return (
            React.DOM.tr(null,
              row.map(function (cell) {
                return React.DOM.td(null, cell);
              })));
        }))));
  }});
```

但是这样太笨拙了，人们还是不接受。

时至今天，人们已经习惯了用JSX来开发APP，回过来想就觉得没什么大不了。毕竟开发效率要紧，赚钱要紧。

JSX只是表现，是一个模板，一个语法糖。它最终还是翻译成React.DOM.xxx堆起来的结构体。它们有一个威风的名字叫虚拟 **DOM** 。像虚拟机，

它不让你再装一台电脑，就可以让你有一台电脑跑两个或3个或10多个操作系统。虚拟DOM也可以让你在不操作DOM的情况下，让你模拟页面在数据发生变动后的大致情形。虚拟DOM是用于预测未来的。然后干什么呢？diff！像git、svn那样比较前后两个版本的不同点。虚拟DOM虽然与真实DOM还是相差十万八千里，如表15-1所示。

表**15-1**

	标签名	属性	子节点集合
虚拟DOM	type	props	children
真实DOM	tagName	attributes	nodeChild

这是一个A标签的虚拟DOM结构。

```
var a = {
  type: 'a',
  props: {
    children: 'React',
    className: 'link',
    href: 'facebook/react · GitHub'
  },
```

```
  _isReactElement: true
}

React.render(a, document.body)
```

图15-12是一个A标签的真实DOM结构。

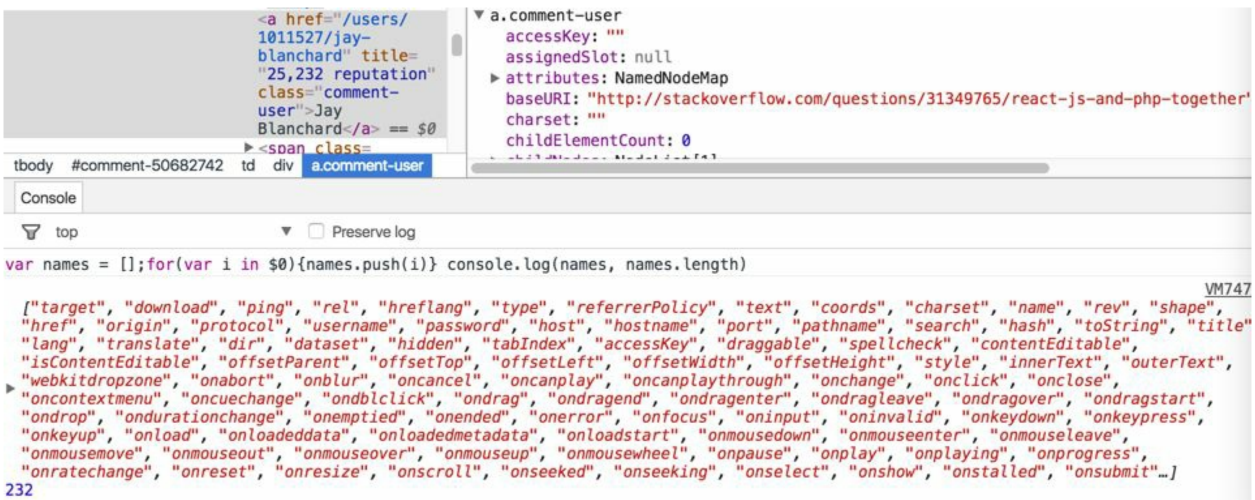


图15-12

属性有230多个，A标签还是比较简单的那种，表单元素input则上升到260多个。此外元素节点有7重原型链。

```
Object.getOwnPropertyDescriptor(HTMLInputElement.prototype, "value")
undefined
Object.getOwnPropertyDescriptor(HTMLInputElement, "value")
undefined
Object.getOwnPropertyDescriptor(HTMLElement, "value")
undefined
Object.getOwnPropertyDescriptor(HTMLElement.prototype, "value")
undefined
Object.getOwnPropertyDescriptor(Element, "value")
undefined
Object.getOwnPropertyDescriptor(Element.prototype, "value")
undefined
Object.getOwnPropertyDescriptor(Node.prototype, "value")
undefined
Object.getOwnPropertyDescriptor(Node, "value")
undefined
```

这些信息，对于我们比较两个DOM是否一致，是没有意义的。因此虚拟DOM就是这么几个属性，但它已经携带了足够多的信息，让它可以描述一个像它这样的DOM是长成怎么样的。

又如文本节点，在虚拟DOM中，它就直接是一个字符串或数字。因为对于文本节点而言，框架只需要知道nodeValue就能创建对应的真实DOM——`document.createTextNode(node Value)`。可见JSX为了减轻虚拟DOM树的内存负担是做了认真的思考。到后来，为了防止每次都找遍历DOM树

查找对应的真实DOM，它还直接将DOM放到虚拟DOM上，大家可以通过getDOMNode方法获取。

此外，虚拟DOM不只是描述P、DIV、FORM这些真实存在的html标签，它还能描述自定义标签。自定义标签与真实标签的区别在于，html肯定是小写，并且框架内部有一个哈希表，将所有html标签都囊括其中。自定义标签则是大写字母开头，它在定义时没有children方法，而是提供了一个render方法。render里面必须返回一个根节点，它可以是其他自定义标签或普通html标签。这带来的好处是，所有标签都是组件，以组件的形式来写代码。

好了，既然我们知道一个DOM长得怎么样，又知道它的虚拟DOM在数据变化后长得怎么样，那么facebook就集中精力改进diff算法，算出如何以最小的步骤刷新视图。

15.4.1　React的diff算法

传统的MVVM是通过访问器属性精确得到要修改的节点，而React每次更新则从组件的根节点开始diff与更新。按理来说，React的更新性能不如传统的MVVM啊，并且diff算法也不是首创，大家都明白这是怎么一回事。计算一棵树形结构转换成另一棵树形结构的最少操作，基本上算法书都已经写明的算法复杂度 $O(n^3)$，其中 n 是树中节点的总数。

$O(n^3)$ 到底有多可怕，这意味着如果要展示1000个节点，就要依次执行上十亿次的比较。这种指数型的性能消耗对于前端渲染场景来说代价太高了！现今的 CPU 每秒钟能执行大约30亿条指令，即便是最高效的实现，也不可能在一秒内计算出差异情况。

因此，如果React只是单纯的引入diff算法而没有任何的优化改进，那么其效率是远远无法满足前

端渲染所要求的性能。通过下面的demo可以清晰的描述传统diff算法的实现过程。

```
// 循环遍历
  for (let i = 0; i < count; i++) {
    const beforeTag = beforeLeaf.children[i];
    const afterTag = afterLeaf.children[i];
    // 添加 afterTag 节点
    if (beforeTag === undefined) {
      result.push({type: "add", element: afterTag});
    // 删除 beforeTag 节点
    } else if (afterTag === void 0) {
      result.push({type: "remove", element: beforeTag});
    // 节点名改变时，删除 beforeTag 节点，添加 afterTag 节点
    } else if (beforeTag.tagName !== afterTag.tagName) {
      result.push({type: "remove", element: beforeTag});
      result.push({type: "add", element: afterTag});
    // 节点不变而内容改变时，改变节点
    } else if (beforeTag.innerHTML !== afterTag.innerHTML) {
      if (beforeTag.children.length === 0) {
        result.push({
          type: "changed",
          beforeElement: beforeTag,
          afterElement: afterTag,
          html: afterTag.innerHTML
      });
      } else {
        // 递归比较
        diffLeafs(beforeTag, afterTag);
      }
    }
  }
  return result;
}
```

因此为了提高diff速度，facebook必须做一些大

胆的创新，将 $O(n^3)$ 复杂度的问题转换成 $O(n)$ 复杂度的问题。

diff策略

- 页面中将一个节点挪到另一个父节点底下的移动操作特别少，因此可以忽略。
- 两个相同组件产生类似的DOM树结构，不同的组件产生不同的DOM树结构。
- 对于同一层次的一组子节点，它们可以通过唯一的id进行区分。

基于以上3个前提策略，React分别对tree diff、component diff以及element diff进行算法优化，事实也证明这3个前提策略是合理且准确的，它保证了整体界面构建的性能。

- tree diff。
- component diff。

- element diff。

扩展阅读如下。

```
http://zhuanlan.zhihu.com/p/20346379
```

15.4.2　React的多端渲染

既然我们可以用DOM来呈现虚拟DOM的外观，我们也可以使用其他介质来呈现它。在2013年，就有一个叫Famo.us的框架，以canvas展现了其高性能。那时Web App一直苦于性能，尽量少用DOM，好了，现在body之内全部是用canvas绘制的，性能哗哗就上去了。很快就有人搞出React canvas，Facebook官方看到有戏，就组织人员开发React Native。

因为有了虚拟DOM这一层，所以通过配备不同的渲染器，就可以将虚拟DOM的内容渲染到不

同的平台。而应用开发者，使用JavaScript就可以通吃各个平台了，如图15-13所示。

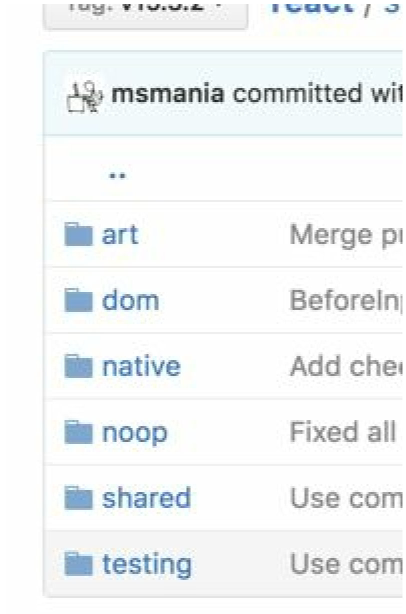


图15-13

React15 添加了SVG的官方渲染。

art是一个旨在多浏览器兼容的Node style CommonJS模块。在它的基础上，Facebook又开发了react-art，封装art，使之可以被react.js所使用，即实现了前端的svg库。然而，考虑到react.js的JSX语法，已经支持将`<cirle> <svg>`等svg

标签直接插入到DOM中（当然此时使用的就不是react-art库了），此外还有HTML canvas的存在，因此，在前端上，react-art并非不可替代。

15.5　性能墙与复杂墙

由于MVVM的便捷性，让前端更容易堆砌庞大的应用，于是它们比以往更加频繁地遇到这两个问题。

性能墙，传统MVVM会将一个普通对象变成一个充满访问器属性的重型对象。如果一个属性是一个对象，那么它也会转换为一个VM重型对象。如果一个属性是一个数组，需要转换为监控数组，即重写其所有方法。对于监控数组的转换，现在也可以用`__proto__`赋予一个新对象就行了。

```
var ap = Array.prototype
var observeArray = {}
Object.getOwnPropertyNames(ap).forEach(function (name) {
    observeArray[name] = ap[name]
})

observeArray.remove = function (el) {
    var index = this.indexOf(el)
    if (index !== -1) {
        this.splice(index, 1)
        return true
    }
    return false
```

```
}
var arr1 = []
//实现数组的子类
arr1.__proto__ = observeArray
arr1.push(11)
console.log(arr1.remove(11))//true
var arr2 = []//这是普通数组
```

如果数组元素是一个个对象，那么它们也会转换为VM。这是一个很大的性能消耗。avalon2现在唯有循环利用这些小VM。这估计等浏览器的版本上去了，我们才可以用Proxy代替Object.defineProperty来实现VM。

在解决工程复杂墙的问题上，业界现在倾向使用组件化来处理，这个下一章专门介绍。但如何优雅地定义组件，这节可以介绍一下。现在流行两种方法，已经挂掉的web component方式与新兴的JSX方式。

web component就是自定义标签，可以说是一堆指令集合，外加生命周期管理。此外web

component带来了slot（插槽）的新概念，解决大段文本或元素片段的传递问题。比如我们创建一个弹出层组件，弹出层中间的内容是怎么传进的呢？slot就很好解决这问题。

下面是弹出层的模板，里面有许多slot元素，它们注定要被替换掉。

```
<div class="modal-mask"  ms-visible="@isShow" ms-effect="{is:'modal'}">
    <div class="modal-box">
        <div class="modal-header">
            <h3>{{@title}}</h3>
            <i class="icon-collapse-alt icon-large modal-close" ms-click="@cbProxy(false)"></i>
        </div>
        <div class="modal-body">
            <slot name="content"></slot>
        </div>
        <div class="modal-footer">
            <button class="btn" ms-click="@cbProxy(false)">取 消</button>
            <button class="btn btn-primary" ms-click="@cbProxy(true)">确 定</button>
        </div>
    </div>
</div>
```

这是使用方式。

```
 <ms-modal :config="@config">
      <p>弹窗的内容</p>
      <p>弹窗的内容</p>
      <p>弹窗的内容结束!</p>
</ms-modal>
```

当各种自定义标签组装起我们的页面时，会是这个样子。不但清新脱俗，还有很强的可维护性，只要瞧一眼它的声明结构就可以清楚地知道它到底要干嘛。

```
<ms-module>
  <ms-chat from="Paul, Addy">
    <ms-discussion>
      <ms-message from="Paul" profile="profile.png" profile="11
8075919496626375791" datetime="2013-07-17T12:02">
        <p>Feelin' this Web Components thing.</p>
        <p>Heard of it?</p>
      </ms-message>
    </ms-discussion>
  </ms-chat>
  <ms-chat>...</ms-chat>
</ms-module>
```

但是，IE6～IE8中是无法识别这样的标签，不过IE支持用冒号隔开的VML标签。标准浏览器也支持带冒号的标签，那么就将所有横杠改成冒号。这

时问题又来了，在Chrome、Firefox、IE11, IE11的IE6兼容模式分别如图15-14、图15-15、图15-16、图15-17所示。

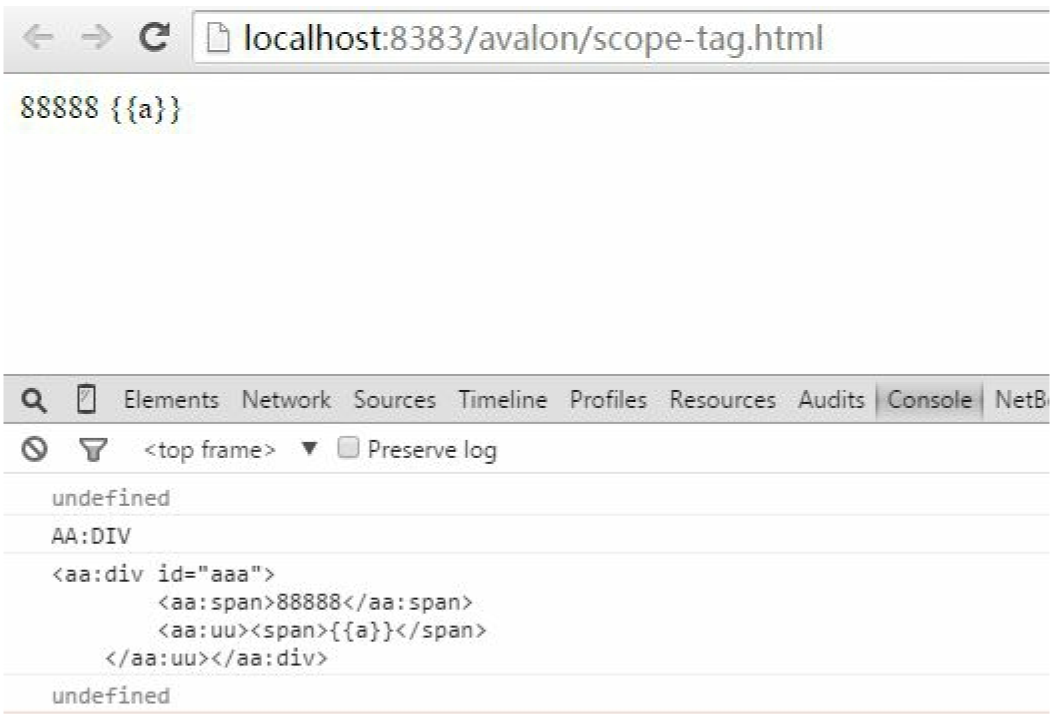


图15-14


localhost:8383/avalon/scope-tag.html
88888 {{a}}
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame> Preserve log
undefined
AA:DIV
<aa:div id="aaa">
<aa:span>88888</aa:span>
<aa:uu>{{a}}
</aa:uu></aa:div>
undefined
88888 {{a}}
控制台 HTML CSS 脚本 DOM 网络 Cookies
清除 保持 概况 全部 错误 警告 消息 调试信息 Cookies
undefined
AA:DIV
<aa:div id="aaa">
<aa:span>88888</aa:span>
<aa:uu>{{a}}
</aa:uu></aa:div>
undefined

图15-15



图15-16



图15-17

我们会发现IE6下实际是多出许多标签，它是

把闭标签也变成一个独立的元素节点，如图15-18所示。

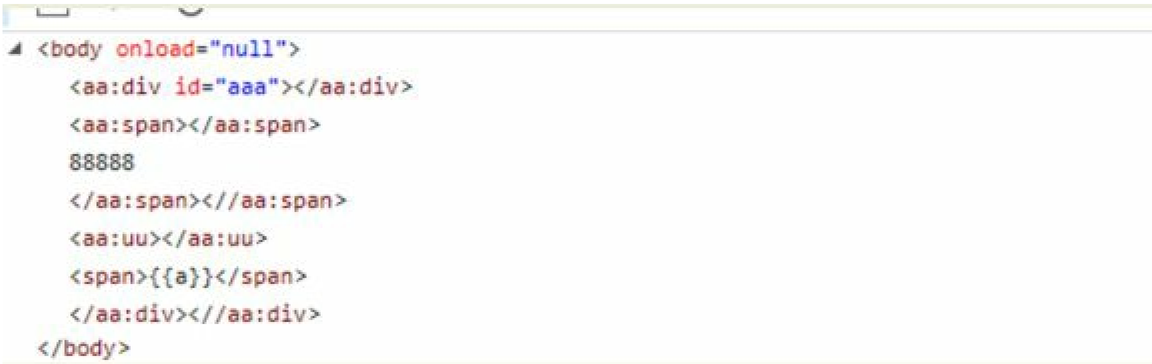


图15-18

缘故是它不能直接使用，需要单独开一个命名空间。

```
if (document.namespaces) {
   var htmlNs = 'http://www.w3.org/1999/xhtml'
   document.namespaces.add('aa',htmlNs)
}
```

由于这兼容性问题，现在业界都倾向将模板放在script与textarea等容器元素里面。使用一些手段，让浏览器不解析里面的文本。这时标签名就可

直接用横杠，或像JSX那样以大小写区分普通标签与自定义标签。为了将里面的标签分解出来，这时就需要一个改造过的特殊html parser。有关html-parser可以搜一下github、avalon2、angular2、ractive.js都内置了html-parser。

笔者收藏了一个迷你的html-parser，大家可以参考一下。

```
https://github.com/lemonde/cms-htmlparser/blob/master/htmlparser.js
```

html-parser能让我们的框架脱离对DOM的依赖，直接产生一堆虚拟DOM，然后这些虚拟DOM也能转换为一个渲染函数，放在后端输入一个完整的页面，解决SEO问题。

有了虚拟DOM，MVVM的流程就变成如图15-19所示。

MVVM的层级也变成这样，最后是VM，接受外界的添加删除修改，中间是虚拟DOM，作为缓存层，预测新的DOM结构与算出最小的修改DOM的步骤，最内层的DOM API与所有要处理的DOM。这其实与我们一贯的软件架构解决方案一样，当我们搞不定某个技术难点，就会进行分治，再介入一层，专门进行它进行攻坚。我们今天讨论争论的MVC、MVP、MVVM等都源自于职能分化和规划的思想与目的，如图15-20所示。

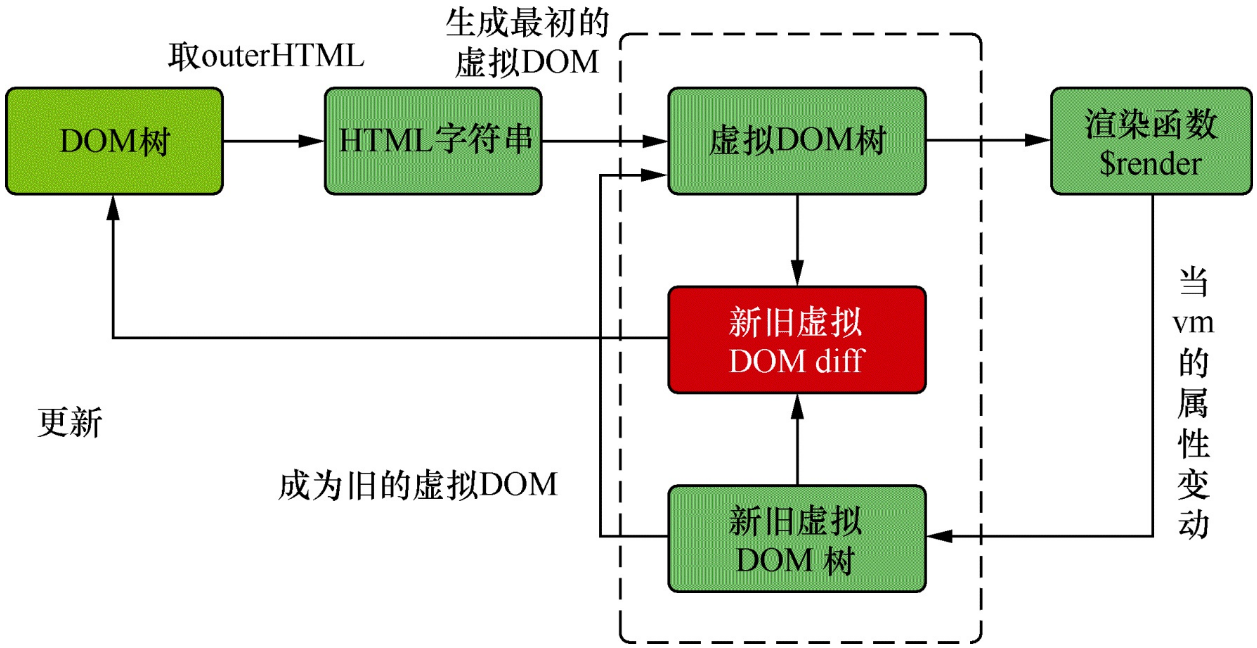

取outerHTML
生成最初的
虚拟DOM
DOM树
HTML字符串
虚拟DOM树
渲染函数
$render
新旧虚拟
DOM diff
当
vm
的
属
性
变
动
更新
成为旧的虚拟DOM
新旧虚拟
DOM 树

图15-19

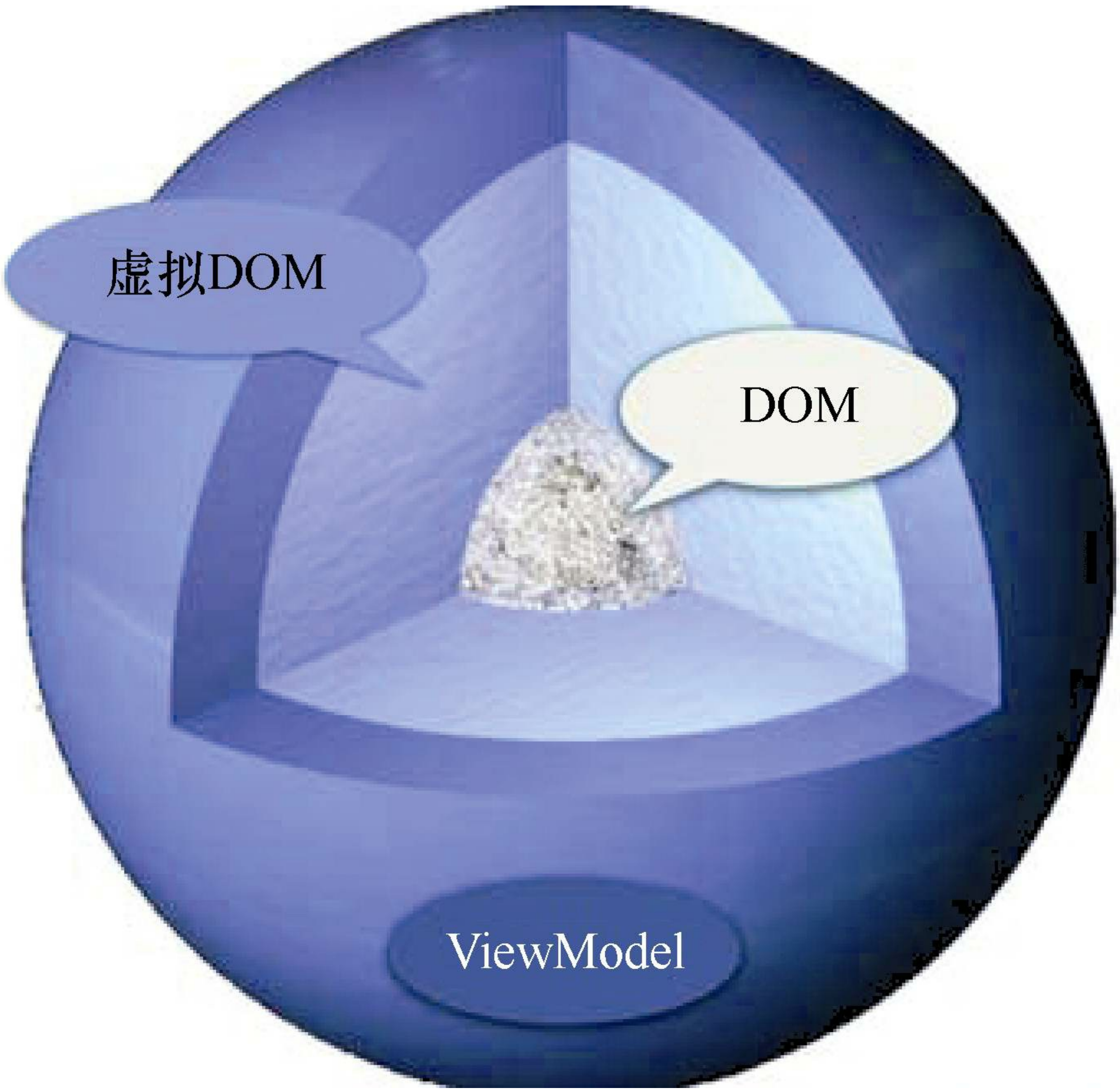
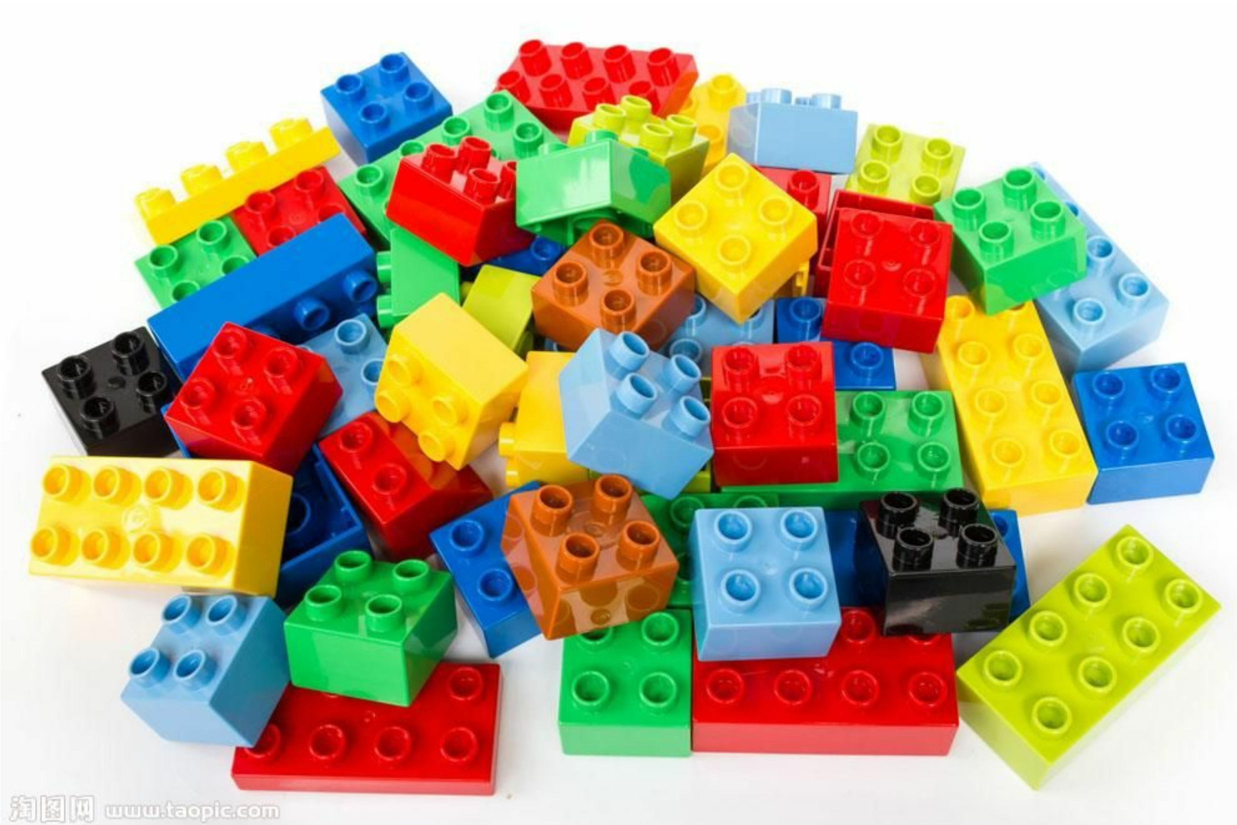


图15-20

参考资料如下。

```
https://www.zhihu.com/question/31809713/
https://www.zhihu.com/question/29504639
https://www.zhihu.com/question/42001493
http://www.cnblogs.com/DebugLZQ/archive/2012/05/15/2501512.html
http://www.jianshu.com/p/9a6845b26856
```

第16章　组件



组件是视图层的分治方案。在Web开发中，我们所做的一切是解决维护成本与代码复用问题。MVVM解决了通用的业务逻辑开发与维护问题，而组件，在上章也指出了，它是复数个指令的集合再

加上生命周期。

16.1　jQuery时代的组件方案

jQuery时代，如果代码组织良好，也是MVC相分离的。M就是插件的默入配置对象与用户的传参对象，V就是用数组形式连结起来的html字符串，jQuery对象允当控制器C。

由于jQuery的API太强大了，动态创建一个节点并且改造它，将它挪到某一处都是轻而易举的事，因此jQuery的组件没有统一的套路，唯一相同点是它都在在jQuery.fn下进行扩展。jQuery.fn也就是jQuery.prototype，定义了一个jQuery.fn.dialog方法，然后使用时，直接就是$(expr).dialog()，是纯命令式的。当然也可以将一大堆字符串变成一个弹出层的样式，然后插入到某元素底下，对于那个时代的人而言，它也是组件。

```
(function($) {//这个东西叫IIFE
    //扩展这个方法到jQuery
    $.fn.extend({
```

```
        //插件名字
        pluginName: function() {
            //遍历匹配元素的集合
            return this.each(function() {
                //在这里编写相应代码进行处理
            });
        }
    });
})()
```

注意，不是所有在jQuery.fn中扩展的方法都是一个组件。jQuery作者的本意是让大家在上面扩展原型方法，它们一般被谦虚地称为插件。插件与组件是不一样的。组件是有UI界面的，插件则没有不一定。或者说，组件是插件的一种特例。

一些优秀的UI库试图对其辖下的组件都进行规范化，并做了一些创新。比如最有名的Bootstrap，带来一项很实用的功能，自动初始化组件！只要用户引入Bootstrap的JS与CSS文件，然后HTML以官网那样组织结构及添加类名，就可以不用写一行JS代码让弹出层、轮播、手风琴等组件出现在你的页面上。

Bootstrap的Dropdown 插件的主体骨架如下。

```
//https://github.com/twitter/bootstrap/blob/master/js/bootstrap
-dropdown.js
!function($) {
    "use strict"; // ECMA262v5的新东西,强制使用更严谨的代码编写
    /* 内部工作的类
     * ========================= */
    var toggle = '[data-toggle=dropdown]';
    var Dropdown = function(element) {
        var $el = $(element).on('click.dropdown.data-api', this
.toggle);
        $('html').on('click.dropdown.data-api', function() {
            $el.parent().removeClass('open');
        });
    };
    Dropdown.prototype = {
        constructor: Dropdown,
        toggle: function(e) {
            /*略*/
        },
        keydown: function(e) {
            /*略*/
        }
    };
    /* 主接口
     * ========================== */
    var old = $.fn.dropdown;
    $.fn.dropdown = function(option) {
        return this.each(function() {
            var $this = $(this),
                    data = $this.data('dropdown');
            if (!data)
                $this.data('dropdown', (data = new Dropdown(thi
s)));
            if (typeof option === 'string')//调用它的实例方法
                data[option].call($this);
        });
    };

    $.fn.dropdown.Constructor = Dropdown;//暴露类名

    /* 无冲突处理
```

```
     * ==================== */
    $.fn.dropdown.noConflict = function() {
        $.fn.dropdown = old;
        return this;
    };
    /*事件代理，智能初始化
     * =================================== */
    $(document)
            .on('click.dropdown.data-api', clearMenus)
            .on('click.dropdown.data-api', '.dropdown form', fu
nction(e) {
        e.stopPropagation();
    })
            .on('click.dropdown-menu', function(e) {
        e.stopPropagation();
    })
            .on('click.dropdown.data-api', toggle, Dropdown.pro
totype.toggle)
            .on('keydown.dropdown.data-api', toggle + ', [role=
menu]',
            Dropdown.prototype.keydown);

}(window.jQuery);
```

我们可以把这两个看作是编写jQuery插件的最佳实践。组件名即新增的原型函数名，内部类使用，默认参数，自动初始化给件。

我们再来看其官方UI库，jQuery ui。在jQuery 1.9中它共有accordion、autocomplete、button、datepicker、dialog、menu、spinner、tabs、slider、

tooltip 10个UI，其中最受欢迎的dateplcker日历组件还没有统一化，其他都是基于$.Widget构建。

```
$.widget("ui.button", {//jquery.ui.button.js
    version: "@VERSION",
    defaultElement: "<button>", //使用什么元素作为它的最外围元素
    options: {
        //默认参数
    },
    _create: function() {//生命周期钩子
        //根据当前元素的情况重置一些参数与绑定事件
    },
    widget: function() {//返回根节点
        return this.buttonElement;
    },
    _destroy: function() {//生命周期钩子
    //移除各种类名，属性与事件
    },
    _setOption: function(key, value) {//如果处理传参
    },
    refresh: function() {//生命周期钩子
        //略
    },
    //略
});
```

如果用过react或avalon的人，就会感叹六七年的UI库已经考虑到根节点、生命周期、数据对象等现在定义组件所必需的要素。由于jQuery操作风格命令式，它不会独立产生一个组件实例让你调用它的方法，那你能过重载一个方法的参数实现组件的

各种操作。不过，jQuery团队最擅长的就是提供优雅的接口，以accordion组件为例。

初始化时传一个对象，方便设置N个配置项。

```
$( ".selector" ).accordion({ heightStyle: "fill" ,{ active: 2 }
});
```

初始化后，第一个参数为字符串“option”时即进入配置模式。如果后面只有一个属性或方法名，那么就是读方法（getter）。如果它们之后还有参数，那就是写方法（setter），作为一个方法的参数或这个属性的新值。

```
// getter
var active = $( ".selector" ).accordion( "option", "active" );
// setter
$( ".selector" ).accordion( "option", "active", 2 );
```

不同的控件会有不同的方法或属性，但由于都是同一个基类，因此会有如下相同的操作。

让控件不可用。它有以下两种操作方式。

第一种，使用配置模式。

```
$( ".selector" ).accordion( "option", "disabled", true );
```

第二种，直接传入“disable”。

```
$( ".selector" ).accordion( "disable" );
```

让控件可用，也对应两种。

第一种，使用配置模式。

```
$( ".selector" ).accordion( "option", "disabled", false);
```

第二种，直接传入“enable”。

```
$( ".selector" ).accordion( "enable" );
```

销毁控件，可传入“destroy”。

```
$( ".selector" ).accordion( "destroy" );
```

如果想得此UI最外围的元素节点的jQuery，可传入“widget”。

```
var widget = $( ".selector" ).accordion( "widget" );
```

至于实现，修改配置与调用方法是很容易的，它们都是走`_setOptions` 方法，问题在于让插件是否可用。一般地，它只对控件的类名下手，添加一个叫做`ui-state-disabled` 的类名，并将options.disable改为false。那么在修改配置时，就无法进入实际操作的那个分支。由于控件必然绑定了许多方法，因此它不是使用jQuery的on、bind、delegate进行绑定，而是使用一个`_on` 的方法。

```
$.Widget.prototype._on = function(suppressDisabledCheck, element, handlers) {
```

```
    var delegateElement,
            instance = this;
    // 第一个参数决定是否检测disabled状态
    if (typeof suppressDisabledCheck !== "boolean") {
        handlers = element;
        element = suppressDisabledCheck;
        suppressDisabledCheck = false;
    }

// 处理参数多态化,可能用户不会传这么多参数,不足部分自己设计补上
    if (!handlers) {
        handlers = element;
        element = this.element;
        delegateElement = this.widget();
    } else {
        element = delegateElement = $(element);
        this.bindings = this.bindings.add(element);
    }
//开始绑定
    $.each(handlers, function(event, handler) {
        function handlerProxy() {
//这里的分支最关键,用于决定用户的操作是否无效化
            if (!suppressDisabledCheck &&
                    (instance.options.disabled === true ||
                            $(this).hasClass("ui-state-disabled
"))) {
                return;
            }
            return (typeof handler === "string" ? instance[ han
dler ] : handler)
                    .apply(instance, arguments);
        }

// 这里就是模拟jQuery核心库proxy方法的实现,加个UUID,方便移除
        if (typeof handler !== "string") {
            handlerProxy.guid = handler.guid =
                    handler.guid || handlerProxy.guid || $.guid
++;
        }

        var match = event.match(/^(\w+)\s*(.*)$/),
                eventName = match[1] + instance.eventNamespace,
                selector = match[2];
        if (selector) {
            delegateElement.delegate(selector, eventName, handl
```

```
erProxy);
        } else {
            element.bind(eventName, handlerProxy);
        }
    });
}
```

有的插件的disable与enable可能非常复杂，需要专门设计一个原型方法。

jQuery通过字符串传参来调用原型方法的设计非常绝妙，基本成为一种套路。

16.2 avalon2的组件方案

从现在的角度来看，jQuery的组件是粗糙的，首先它连独立性的问题也没有解决。组件只是jQuery某个方法在某种传参下产生的现象。组件涉及的DOM也乱七八糟地糅杂业务逻辑中，没有统一定义的DOM，也就没有统一设计的样式规划。当然，若干年后，再看我们这些引以为豪的组件，或许也不过尔尔，挑出一大堆毛病。用工业体系的发展史来对比，jQuery组件就像蒸汽机发明之前的早期机械，如《木兰辞》里面的机杼，自动化程度非常低。

avalon2是产生于现在已经死去的Web Components标准之后，此时CSS已经大规模被更取替。SASS带来CSS的模块化与组件化。有时候，组件不一定需要JavaScript，有一堆html放在那里，CSS让它呈现成一个弹出层，那么用户就认为它是

一个弹出层。SASS因为了有了变量，有了函数，因此实现换肤功能易如反掌。

再看Web Components标准留下的遗产。

（1）`template` 模板元素，为我们提供了一个比script标签更好的容器元素，并且它也是一个更好的天然html parser。在过去，我们使用innerHTML来做html parser，需要打许多补丁。

（2）Shadow DOM 可以理解为一份有独立作用域的html片段。这些html片段的css环境和主文档隔离的，各自保持内部的独立性。也是Shadow DOM的独特性，使得组件化成为了可能。

（3）自定义标签，能让我们优雅地声明一个组件，这个组件拥有完整的生命周期钩子，可以通过它们与其他组件联动。

（4）import机制解决组件模板的加载问题。

有人进一步将它们抽象成设计的组件5大规范，如图16-1所示。

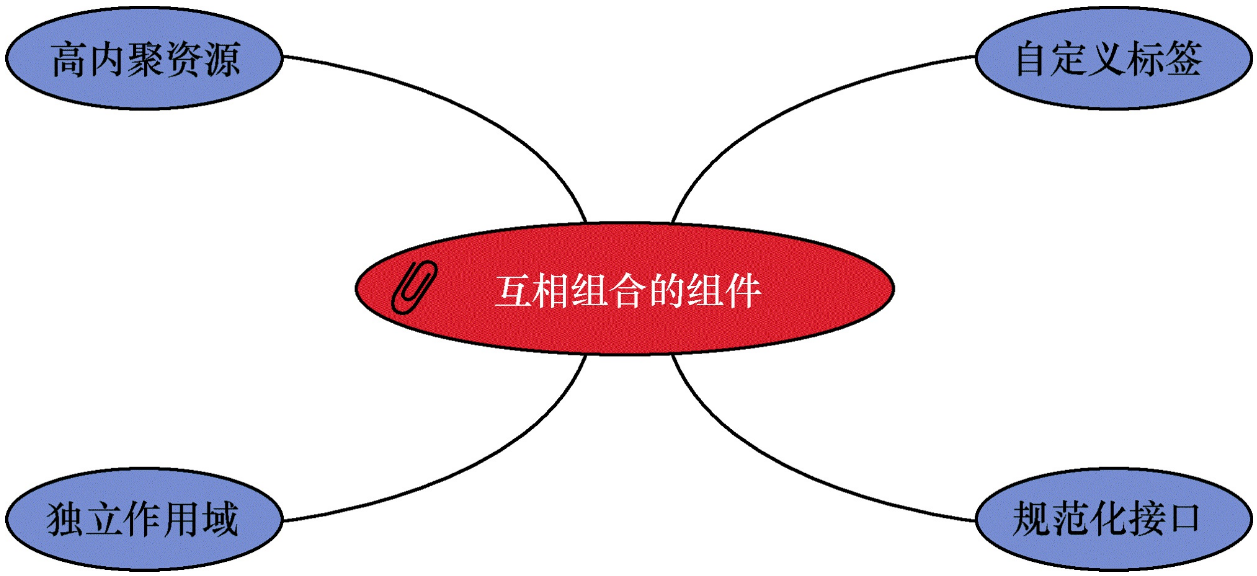


图16-1

- 高内聚资源——组件资源内部高内聚，组件资源由自身加载控制。
- 独立作用域——内部结构密封，不与全局或其他组件产生影响。
- 自定义标签——定义组件的使用方式。
- 规范化接口——组件接口有统一规范，或者是生命周期的管理。
- 可相互组合——组件真正强大的地方，组件间组装整合。

avalon2正是遵循这些规范重新设计了其组件系统。

（1）组件基于ES6 import 或common.js构建，每个组件都有自己的JS文件、模板文件与SASS文件，实现结构行为样式分离。当然如果模板太小，只有一行也可以放进JS文件。有的组件就只需要默认样子或使用整体引用的CSS库，那也可以略去SASS文件。

（2）组件是一个独立的VM。

（3）使用时，是一个特殊的元素节点（针对IE6～IE8）或自定义标签。

（4）组件的模板内可以使用其他组件的标签（强制依赖其他组件），或者在使用组件时，通过slot机制，在里面引用其他组件的标签（自由使用使用其他组件）。比如说弹出层，是存在确认取消

按钮，那么它是强制依赖button组件，但它的内部是什么是用户决定，用户可以在里面使用面板、拖动条、日历，这是自由搭配。

（5）组件都有相同的生命周期钩子函数，onInit、onReady、onViewChange、onDispose。

下面是一个button组件的定义。

```
avalon.component('ms-button', {
    template: '<button type="button"><span><slot /></span></but
ton>',
    defaults: {
        buttonText: "button"
    },
    soleSlot: 'buttonText'
})
```

然后使用时是这样用的。

```
<ms-button>你好<ms-button>
```

最后在页面上生成这样的结构，如图16-2所

示。

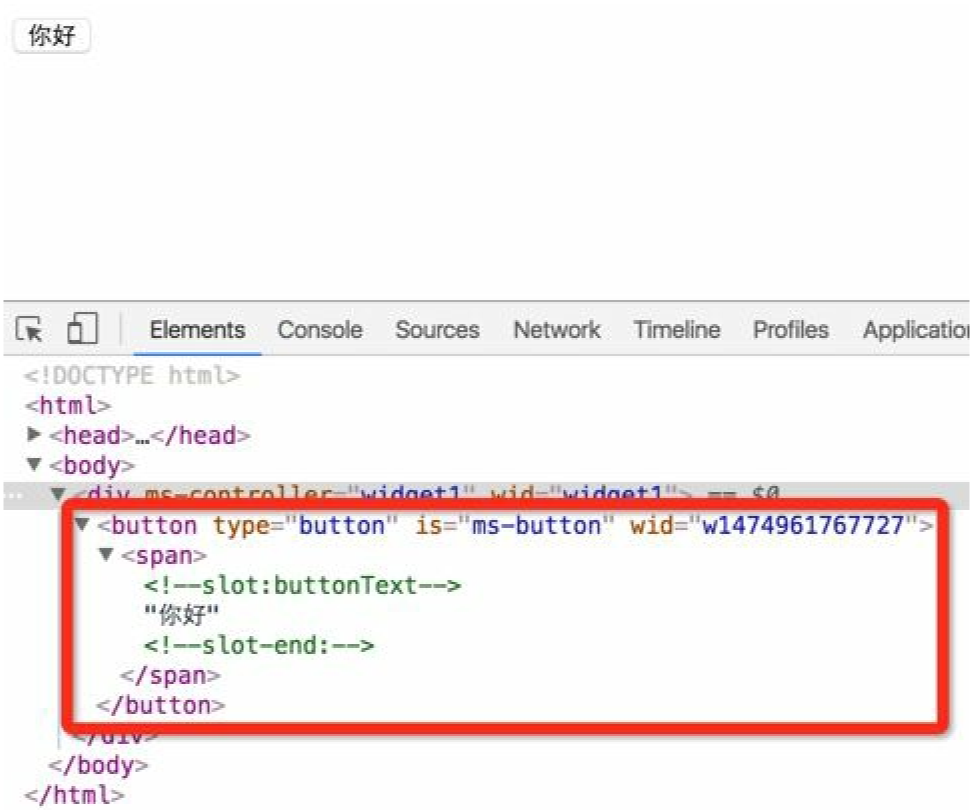


图16-2

但不是所有浏览器都支持自定义标签，在上一章中，我们也提出了一个解决之道，但还有更好的办法。这次，我们引入了组件容器的概念。为了兼容IE6～IE8，我们不得不做出妥协，不只有自定义标签可以成为组件，某些特殊元素也可以成为组

件。

16.2.1　组件容器

组件容器是一个占位用的元素节点，当avalon扫描到此位置上时将它替换成组件。

在avalon2中有4类标签可以用作组件容器，分别是wbr、xmp、template，及ms-开头的自定义标签。

其兼容性如表16-1所示。

表**16-1**

元素	类型	说明
wbr	所有浏览器，自闭合标签	需要使用ms-widget来指定组件类型
xmp	所有浏览器，闭合标签	需要使用ms-widget来指定组件类型，里面可以使用slot属性元素
template	IE9+及W3C浏览器，闭合标签	需要使用ms-widget来指定组件类型，里面可以使用slot属性元素

ms-*	IE9+及W3C浏览器，闭合标签	可以省略ms-widget，里面可以使用slot属性元素

闭合标签，比如`<div></div><a></a>`。

自闭合标签，比如`<input><img><br><link>`。

为什么要选用wbr、xmp、template这3种标签呢？xmp、template我们在节点模块的容器元素已经解析过，它们在内部只是一个文本节点或没有节点，不会增加DOM树的整体节点数，是天然模板容器。而wbr，它是用于某些不用定义内容的组件，它纯粹做占位与被替换掉。

根据表16-1所示，我们可以划分它们的适用场合。如果要兼容IE6～IE8，那么只能使用wbr、xmp来做组件容器；如果不打算支持IE，那么使用template元素与自定义标签。

这里需要说明一下，自定义标签其实在IE9～

IE11中也不被支持，甚至许多浏览器也不支持，它都被浏览器当成一种HTMLUnknownElement的元素实例，它有许多限制。首先它是闭合标签，不能只写一半，其次它不支持ID，无法被`document.getElementById`索引到，最后它也不支持绑定动画结束事件（如onnimationend与ontransitionend）。

自定义标签从标签名可以很好地声明它是什么组件，可以xmp、template、wbr需要使用其他属性标识一下。早前流行的polymer库是用is属性来声明，这样受众面比较广，因此avalon2也借鉴过来。

因此声明一个button组件，在avalon中有4种方式。

```
<xmp is="ms-button">按钮</xmp>
<template is="ms-button">按钮</template>
<wbr is="ms-button" />
<ms-button>按钮</ms-button>
```

至于自定义标签一定要用ms-开头呢？这其实也是人为的规定。有句话说得好，架构是对客观不足的妥协，规范是对主观不足的妥协 。架构师的工作，基本上就是不停地在各种各样的矛盾中正确地取舍。实现架构设计的优雅，与特别紧迫的需求，也是一个矛盾，怎么样控制节奏，在某些时候做一些妥协。不要忘记我们的初衷，一切为业务服务，为了保证需求被实现，我们允许存在一些瑕疵。

因为IE6～IE8不支持自定义标签，我们需要引进组件容器的概念，让更多标签可以转换为组件。为了防止影响所有自定义标签（因为我们难免使用第三方库，假如它们也用了自定义标签来进行某种业务操作，这就有互相影响之虞），我们要有选择地对某些前缀的自定义标签进行组件化。

16.2.2　配置对象

配置对象是解决组件的传参问题。在avalon2中，是通过ms-widget属性来添加额外参数。在avalon2的组件定义中，已经有一个defaults配置对象，它规定了组件的默认行为与外观。但防止千篇一律，于是有了传参的问题。由于组件已经标签化，于是只能通过属性来作为传参，有的框架规定自定义标签上所有属性，包括id什么，都是组件的传参，有的框架则规定用某个属性来传进所有用到的参数，avalon2就是后者。这优势在于元素的属性名都会被浏览器小写化，使用第一种风格，你无法传入一些有大写字母的属性。

```
<ms-datepicker :widget="{'number-of-months':3}" ></ms-datepicker>
```

16.2.3　slot机制

配置对象，只能添加一些很简短的属性，但如果要传入一些很长的字符串就不美观不方便了。当

然我们也可以将它先定义了VM中的一个属性，然后再在widget对象中引用，但这样一来就不直观。于是有了slot机制。slot机制是web components规范发明的，在上一章也简单介绍了它。

为了深入理解它，我们需要引入更多概念：插槽元素 与插卡元素 。插槽元素是用占位与替换，插卡元素是用来替换别人的。更直接地说，插槽元素是定义在template中，而插卡元素是定义组件容器的内部，是用户传参的一部分。

我们看一下ms-view 组件的定义与使用。

```
avalon.component('ms-view',{
    template:"<div class="view"><slot name="content" /></div>",
    defaults: {
       content: ""
    }
})
<div ms-controller='test'>
<ms-view>
<div slot="content">这是子视图的内容</div>
</ms-view>
</div>
```

这上面的<slot name="content" />就是插槽元素，×××就是插卡元素。

```
<div class="view">
<!--slot:content-->
<!--slot-end:-->
</div>
```

组件容器中带slot属性的元素，<div slot="content">这是子视图的内容</div>，就是插卡元素。

想想你的电脑主板上的各种插槽，有插CPU的，有插显卡的，有插内存的，有插硬盘的，所以假设有个组件是computer，其模板则如下。

```
<div class="computer">
  <slot name="CPU">这儿插你的CPU</slot>
  <slot name="GPU">这儿插你的显卡</slot>
  <slot name="Memory">这儿插你的内存</slot>
  <slot name="Hard-drive">这儿插你的硬盘</slot>
</div>
```

那么你要攒一个牛逼轰轰的电脑，就可以这么写。

```
<ms-computer>
  <div slot="CPU">Intel Core i7</div>
  <div slot="GPU">GTX980Ti</div>
  <div slot="Memory">Kingston 32G</div>
  <div slot="Hard-drive">Samsung SSD 1T</divt>
</ms-computer>
```

此外，插槽元素也可以是其他自定义标签。

```
<ms-computer>
  <ms-cpu slot="CPU">Intel Core i7</ms-cpu>
  <ms-gpu slot="GPU">GTX980Ti</gpu>
  <ms-memory slot="Memory">Kingston 32G</ms-memory>
  <ms-ddr slot="Hard-drive">Samsung SSD 1T</ms-ddr>
</ms-computer>
```

插槽机制可以解决我们传入大片内容的难题，多个slot元素拥有同一个name值，如图16-3所示。

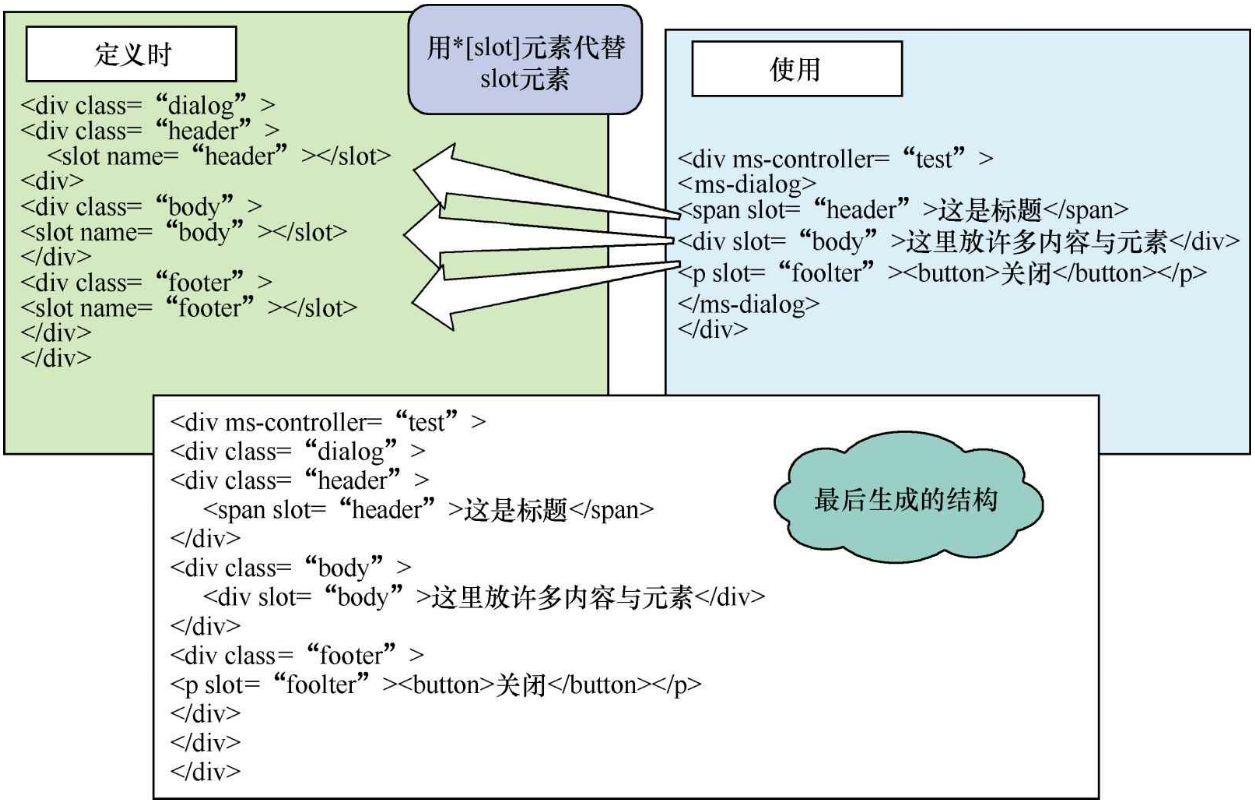


图16-3

16.2.4　soleSlot机制

中文叫单插槽 或匿名插槽 。这是插槽机制的一个特例。

例如我们做一个按钮组件。

```
avalon.component('ms-button', {
    template: '<button type="button"><span><slot name="buttonTe
xt" /></span></button>',
    defaults: {
```

```
        buttonText: "click me"
    }
})
```

那么外面要这么使用。

```
<ms-button><b slot="buttonText">这是按钮</b></ms-button>
```

事实上我们只想传入一个文本，不想再包一个什么b元素，这样定义太冗余了。就像button标签，可以直接这样写。

```
<button>按钮</button>
```

于是就有了单插槽机制。它要求组件内部只有一个地方可以插入东西，并且将组件容器的所有孩子或文本都作为一个插卡。

我们看一下新的定义与声明方式。

```
avalon.component('ms-button', {
```

```
    template: '<button type="button"><span><slot /></span></but
ton>',
    defaults: {
          buttonText: "click me"
    },
    soleSlot: 'buttonText'
})
<ms-button>xxx</ms-button>
```

是不是简单多了。我们改正一下compute的例子。

```
avalon.component('ms-button', {
    template: '<div class="computer"><slot /></div>',
    defaults: {
        configure: "各种配置"
    },
    soleSlot: 'configure'
})

<ms-computer>
  <ms-cpu>Intel Core i7</ms-cpu>
  <ms-gpu>GTX980Ti</gpu>
  <ms-memory>Kingston 32G</ms-memory>
  <ms-ddr>Samsung SSD 1T</ms-ddr>
</ms-computer>
```

16.2.5 生命周期

avalon2组件拥有完善的生命周期钩子，方便大家做各种操作。

onInit，这是组件的vm创建完毕就立即调用时，它对应的元素节点或虚拟DOM都不存在。只有当这个组件里面不存在子组件或子组件的构造器都加载回来时，它才开始创建其虚拟DOM，否则原位置上被一个注释节点占着。

onReady，当其虚拟DOM构建完毕时，它就生成其真实DOM，并用它插入到DOM树，替换掉那个注释节点。相当于其他框架的attachedCallback、inserted、componentDidMount。

onViewChange，当这个组件或其子孙节点的某些属性值或文本内容发生变化时，就会触发它。它比Web Component的attributeChangedCallback更加给力。

onDispose，当这个组件的元素被移出DOM树时，就会执行此回调，它会移除相应的事件，数据与vmodel。

具体用法，可以参看这里。

```
https://github.com/RubyLouvre/avalon/blob/2.1.6/perf/component/
%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F.html
```

avalon2的组件化方案可谓是现时流行的组件标签化一种体现与追随。

16.3 React的组件方案

React的组件方案就是JSX，它会自动帮你React.createClass、React.createElement、React.Component……因此不用JSX，光是区分这些构造器就把你搞晕。并且在长期的快速迭代中，React的组件化方案就在微调中，之前的最佳实践就沦为最糟实践。

16.3.1 React组件的各种定义方式

1．React.createClass

这是React定义组件最常见的方式，创建React组件对应的类，描述你将要创建组件的各种行为，其中只有当组件被渲染时需要输出的内容的render接口是必须实现的，其他都是可选。

```
var Timer = React.createClass({
  render: function() {
    return <reactNode> <span>test</span> <span>test</span> </re
```

```
actNode>;
  }
});

ReactDOM.render(<Timer />, app);
```

由于这段代码是需要经过编译的，因此<Timer/>能找到自己对应的构造器，否则需要指定**displayName** 属性。

```
 React.createClass({
 displayName: "MyComponent"
  render: function() {
    return <reactNode> <span>test</span> <span>test</span> </re
actNode>;
  }
});

ReactDOM.render(<MyComponent />, app);
```

2. React.createElement

创建React组件实例（虚拟DOM），支持type、props、children 3个参数。

```
ReactElement.createElement = function(type, props, children) {
  //....
}
```

比如上面的<MyComponent />，用JSX就是React.createElement(MyComponent)。

3. React.createFactory

通过工厂方法创建React组件实例，在JS里要实现工厂方法只需创建一个带type参数的createElement的绑定函数。

```
ReactElement.createFactory = function(type) {
  var factory = ReactElement.createElement.bind(null, type);
  return factory;
};
```

创建模式的目的是隔离与简化创建组件的过程，模式的东西自然是可用可不用，如果需要批量创建某个组件时，可以通过工厂方法来实现。

```
var div = React.createFactory('div');
var root = div({ className: 'my-div' });
React.render(root, document.getElementById('example'));
```

React.DOM.div、React.DOM.span 等都是预先定义好的“Factory”。“Factory”用于创建特定“ReactClass”的“Element”。

4．es6 Class Component

从 React 0.13 开始，可以使用ES6 Class Component 代替 React.createClass 了。

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

React.Component 是基类，通过extends来创建它的一个子类。

React.createClass和extends Component的区别主要有4个方面。

（1）语法区别。

（2）propType和getDefaultProps。

（3）状态的区别。

（4）this区别。

1. 语法区别

React.createClass

```
import React from 'react';
const Contacts = React.createClass({
  getInitialState: function(props){
     return {count: props.initialCount}
  },
  tick: function() {
    this.setState({count: this.state.count + 1});
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default Contacts;
```

2. React.Component

```
import React from 'react';
```

```
class Contacts extends React.Component {
//constructor方法是可选的，React.createClass 中的某些工作，可以直接在ES6 Class 的构造函数中来//完成，例如：getInitialState 的工作可以被构造函数所替代
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default Contacts;
```

3. propType和getDefaultProps

React.createClass：通过proTypes对象和getDefaultProps()方法来设置和获取props。

```
import React from 'react';

const Contacts = React.createClass({
  propTypes: {
    name: React.PropTypes.string
  },
  getDefaultProps() {
    return {

    };
  },
  render() {
```

```
    return (
      <div></div>
    );
  }
});

export default Contacts;
```

React.Component：设置两个属性propTypes和defaultProps。

```
import React form 'react';
class TodoItem extends React.Component{
    static propTypes = { // as static property
        name: React.PropTypes.string
    };
    static defaultProps = { // as static property
        name: ''
    };
    constructor(props){
        super(props)
    }
    render(){
        return <div></div>
    }
}
```

4. 状态的区别

React.createClass：通过getInitialState()方法返回一个包含初始值的对象。

React.Component：通过constructor设置一个state对象。

5. this区别

React.createClass：会正确绑定this。

```
import React from 'react';

const Contacts = React.createClass({
  handleClick() {
    console.log(this); // React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>//会切换到正确的this上
下文
    );
  }
});

export default Contacts;
```

React.Component：由于使用了ES6，这里会有些微不同，属性并不会自动绑定到React类的实例上。

```
import React from 'react';
class TodoItem extends React.Component{
    constructor(props){
```

```
        super(props);
    }
    handleClick(){
        console.log(this); // null
    }
    handleFocus(){  // manually bind this
        console.log(this); // React Component Instance
    }
    handleBlur: ()=>{  // use arrow function
        console.log(this); // React Component Instance
    }
    render(){
        return <input onClick={this.handleClick}
                              onFocus={this.handleFocus.bind(th
is)}
                              onBlur={this.handleBlur}/>
    }
}
```

当然，我们可以在组件初始化时，直接对其所有事件回调进行劫持，强制bind(this)。

```
import React from 'react';

class Contacts extends React.Component {
  constructor(props) {
     super(props);
     Object.getOwnPropertyNames(this).
     forEach(function(name){
         if(typeof this[name] === 'function'){
               this[name] = this[name].bind(this)
         }
     }, this)
  }
  handleClick() {
    console.log(this); // React Component instance
  }
  render() {
```

```
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default Contacts;
```

6. Stateless Function Components

到了React 0.14，又提倡这种新的组件方式，不过夹在它们中间还有一种叫Pure Components 。

```
import React, { PureComponent } from 'react'

class Text extends PureComponent {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

而Stateless Components就是去掉extends与constructor的Pure Component，也叫Stateless Function。可见自从React引进了ES6后，致力于创建更短凑有力的组件声明方式。

```
const Text = (props) =>
```

```
    <p>{props.children}</p>
```

或者改成ES5风格。

```
var Text = function (props) {
    return <div>{props.children}</div>
}
```

这种无状态函数式组件的写法也是支持设置默认的Props类型与值的。

```
const Text = ({ children }) =>
  <p>{children}</p>
Text.propTypes = { children: React.PropTypes.string };
Text.defaultProps = { children: 'Hello World!' };
```

我们也可以利用ES6默认函数参数的方式来设置默认值。

```
const Text =  ({ children = 'Hello World!' })  =>
  <p>{children}</p>
```

函数式组件中并不需要进行生命周期的管理与状态管理，因此React并不需要进行某些特定的检查或者内存分配，从而保证了更好地性能表现。

7. Higher Order Components

随着React组件被函数化，那么它就可以使用函数式编程里的某些东西，比如高阶函数，于是产生了高阶组件。高阶函数是返回一个新的函数，而高阶组件是返回一个新的组件。

```
const connect = (mapStateFromStore) => (WrappedComponent) => {
  class InnerComponent extends Component {

    static contextTypes = {
      store: T.object
    }

    state = {
      others: {}
    }

    componentDidMount () {
      const { store } = this.context
      this.unSubscribe = store.subscribe(() => {
        this.setState({ others: mapStateFromStore(store.getStat
e()) }
      })
    }

    componentWillUnmount () {
      this.unSubscribe()
```

```
    }

    render () {
      const { others } = this.state
      const props = {
        ...this.props,
        ...others
      }
      return <WrappedComponent {...props} />
    }
  }

  return InnerComponent
}
```

它们的结构大都是这样的。

```
config => {
    return Component=> {
        return HighOrderCompoent
    }
}
```

通过高阶组件来创建新组件比通过继承创建子组件，其维护性更好，这也是编程界的共识（组合优于继承）。但高价组件是社区发展的产物，没有统一创建方案，因此存在一些隐患。

```
https://segmentfault.com/a/1190000004034179
```

大致介绍了这么多创建组件的方式后，大家可能已经眼花缭乱，心中会冒出，React真是一个“奇葩”啊。JSX已经特立独行，还带来这一个奇怪的组件形式。

16.3.2　React组件的生命周期

React组件拥有丰富的生命周期钩子函数，遍布它的各个阶段。其生命周期可被分为挂载（Mounting）、更新（Updating）和卸载（UnMounting）3个阶段。

1．挂载阶段

这是React组件生命周期的第一个阶段，也可以称为组件出生阶段，这个阶段组件被初始化，获得初始的props并定义将会用到的state。此阶段结束时，组件及其子元素都会在UI中被渲染（DOM，

UIview等），我们还可以对渲染后的组件进行进一步的加工。这个阶段的所有方法在组件生命中只会被触发一次。

- getDefaultPropos：只调用一次，实力之间共享引用。
- getInitialState：初始化每个实例特有的状态。
- componentWillMount：render之前最后一次修改状态的机会。
- render：只能访问this.props和this.state，只有一个顶层组件，不允许修改状态和DOM输出。
- componentDidMount：成功render并渲染完成真实DOM后触发，可以修改DOM。

2. 更新阶段

这个阶段的函数会在组件的整个生命周期中不断被触发，这是组件一生中最长的时期。这个阶段的函数可以获得新的props，可以更改state，可以对

用户的交互进行反应。

- componentWillReceiveProps:父组件修改属性触发，可以修改新属性，修改状态。
- shouldComponentUpdate:返回false会阻止render调用。
- componentWillUpeate:不能修改属性和状态。
- render:只能访问this.props和this.state，只有一个顶层组件，不允许修改状态和DOM输出。
- componentDidUpdate:可以修改DOM。

3. 卸载阶段

这是组件生命的最后一个阶段，也可以被称为是组件的死亡阶段，此阶段对应组件从Native UI中卸载之时，具体说来可能是用户切换了页面，或者页面改变去除了某个组件，卸载阶段的函数只会被触发一次，然后该组件就会被加入浏览器的垃圾回收机制。

- componentWillUnMount：在删除组件之前进行清理操作，比如计时器和事件监听器（见图16-4）。

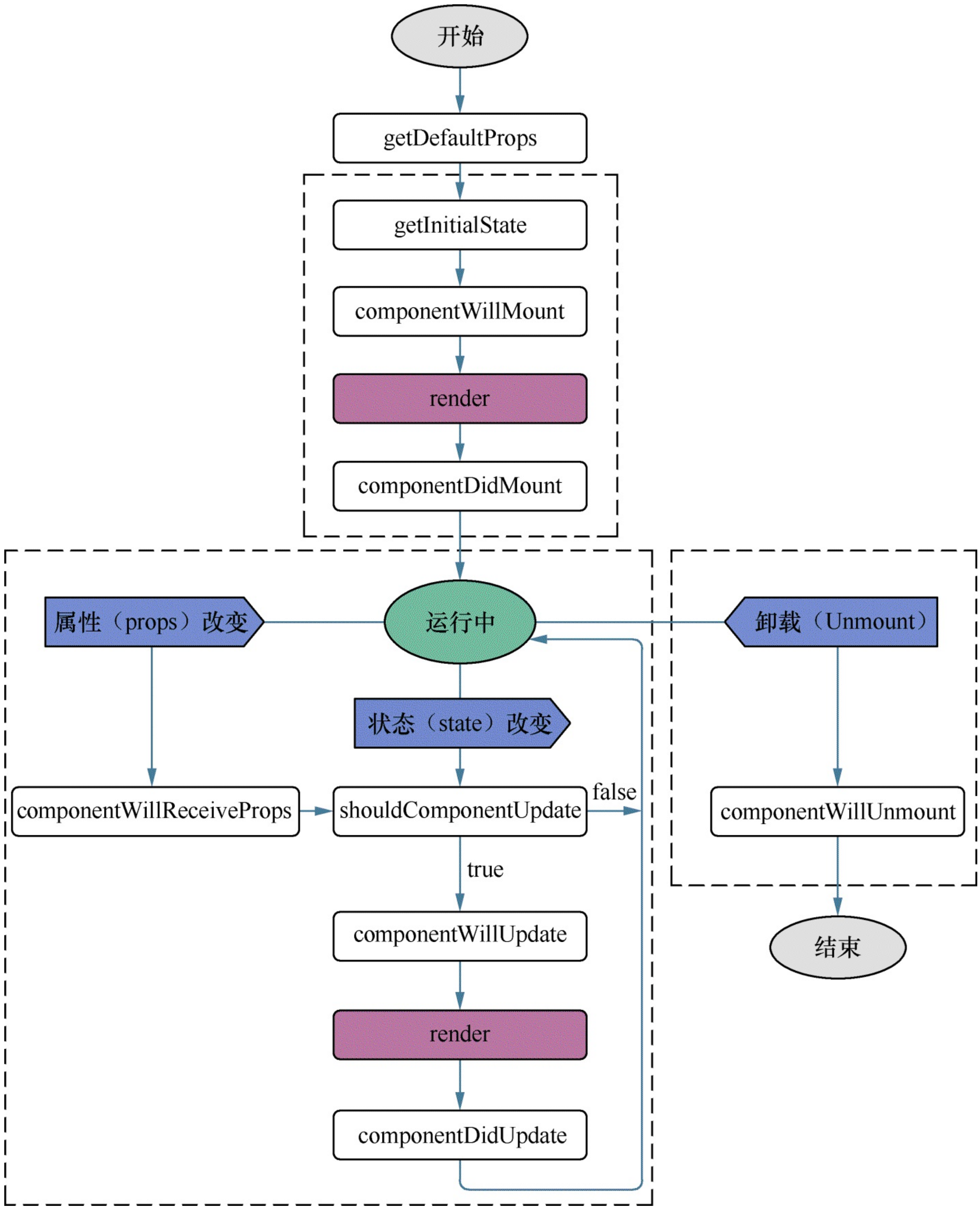

开始
getDefaultProps
getInitialState
componentWillMount
render
componentDidMount
属性（props）改变
运行中
卸载（Unmount）
状态（state）改变
componentWillReceiveProps
shouldComponentUpdate
false
componentWillUnmount
true
componentWillUpdate
结束
render
componentDidUpdate

图16-4

16.3.3　React组件间通信

React推崇数据单向流动，都是从父组件流向子组件。经过大家的不懈努力，把各种关系的组件通信方式总结出来。

父组件->子组件：props。

子组件->父组件：callback。

子组件->子组件：子组件通过回调改变父组件中的状态，通过props再修改另一个组件的状态。

1．父子组件间通信

```
//CalendarControl与CalendarHeader
var CalendarControl = React.createClass({
    getDefaultProps: function () {
        var newDate = new Date();
        return {
            year: util.formatDate(newDate, 'yyyy'),
            month: parseInt(util.formatDate(newDate, 'MM')),
            day: parseInt(util.formatDate(newDate, 'dd'))
        };
    },
    render: function () {
        return (
            <div>
                <CalendarHeader year="this.props.year" month="t
his.props.month" day="this. props.day"/>
```

```
            </div>
        )
    }
});
```

2. 子父组件间通信

```
var CalendarControl = React.createClass({
    getInitialState: function () {
        var newDate = new Date();
        return {
            year: util.formatDate(newDate, 'yyyy'),
            month: parseInt(util.formatDate(newDate, 'MM')),
            day: parseInt(util.formatDate(newDate, 'dd'))
        };
    },
    //给子组件一个回调函数，用来更新父组件的状态，然后影响另一个组件
    handleFilterUpdate: function (filterYear, filterMonth) {
        this.setState({
            year: filterYear,
            month: filterMonth
        });
    },
    render: function () {
        return (
            <div>
                <CalendarHeader updateFilter={this.handleFilter
Update}/>
            </div>
        )
    }
});

var CalendarHeader = React.createClass({
    getInitialState: function () {
        var newDate = new Date();
        return {
            year: util.formatDate(newDate, 'yyyy'),//设置默认年为
今年
            month: parseInt(util.formatDate(newDate, 'MM'))//设
置默认日为今天
```

```
        };
    },
    handleLeftClick: function () {
        var newMonth = parseInt(this.state.month) - 1;
        var year = this.state.year;
        if (newMonth < 1) {
            year--;
            newMonth = 12;
        }
        this.state.month = newMonth;
        this.state.year = year;
        this.setState(this.state);//在设置了state之后需要调用setSta
te方法来修改状态值，
        //每次修改之后都会自动调用this.render方法，再次渲染组件
        this.props.updateFilter(year, newMonth);
    },
    handleRightClick: function () {
        var newMonth = parseInt(this.state.month) + 1;
        var year = this.state.year;
        if (newMonth > 12) {
            year++;
            newMonth = 1;
        }
        this.state.month = newMonth;
        this.state.year = year;
        this.setState(this.state);//在设置了state之后需要调用setSta
te方法来修改状态值，
        //每次修改之后都会自动调用this.render方法，再次渲染组件，以此向
父组件通信
        this.props.updateFilter(year,newMonth);
    },

    render: function () {
        return (
            <div className="headerborder">
                <p>{this.state.month}月</p>
                <p>{this.state.year}年</p>
                <p className="triangle-left" onClick={this.hand
leLeftClick}> </p>
                <p className="triangle-right" onClick={this.han
dleRightClick}> </p>
            </div>
        )
    }
});
```

3. 兄弟组件间通信

```
var CalendarControl = React.createClass({
    getInitialState: function () {
        var newDate = new Date();
        return {
            year: util.formatDate(newDate, 'yyyy'),
            month: parseInt(util.formatDate(newDate, 'MM')),
            day: parseInt(util.formatDate(newDate, 'dd'))
        };
    },
    //给子组件一个回调函数，用来更新父组件的状态，然后影响另一个组件
    handleFilterUpdate: function (filterYear, filterMonth) {
        this.setState({
            year: filterYear,
            month: filterMonth
        });//刷新父组件状态
    },
    render: function () {
        return (
            <div>
                <CalendarHeader updateFilter={this.handleFilter
Update}/>
                <CalendarBody
                    year={this.state.year}
                    month={this.state.month}
                    day={this.state.day}
                />//父组件状态被另一个子组件刷新后，这个子组件就会被刷新
            </div>
        )
    }
});
```

4. Flux

不过如果是一个相隔很远的组件进行通信，虽然也能找到最近的共同组件进行传递，但性能是个问题。这时就要介绍subpub模式的EventBus了。Fackbook自家的解决方案是Flux。Flux应用主要分为4个主要的部分：Views、Actions、Dispatcher、Stores（见图16-5和表16-2）。

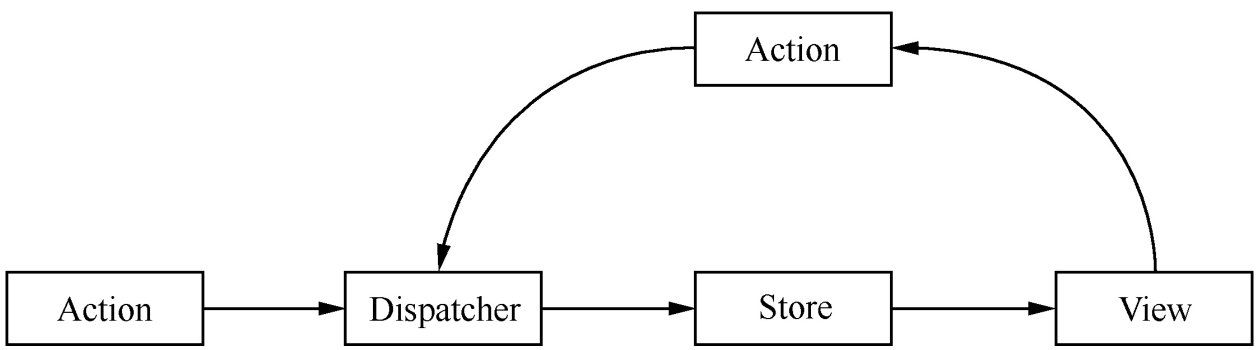


图16-5

表**16-2**

Name	**Description**
View	视图层，React组件
Actions	行为动作层，可以看成是修改Store的行为抽象；
Dispatcher	管理着应用的数据流，可以看为Action到Store的分发器

Stores	管理着整个应用的状态和逻辑，类似MVC中的Model

有关Flux的其他介绍自己用百度或谷歌查找。由于Facebook只是给出Flux的架构图与接口，至于怎么实现，它自家搞了好久，因此存在众多实现问题。在这些混乱中，最终被Redux超越了。

5．Redux

按照Redux官方的描述Redux is a predictable state container for JavaScript apps.，其中predictable和state container体现了它的作用。那么如何来理解可预测化的呢？这里会有一些函数式编程方面的思想，在Redux中reducer函数是一个纯函数，相同输入一定会是一致的输出，所以确定输入的state那么reducer函数输出的state一定是可以被预测的，因为它只会进行单纯的计算，保证正确的输出。状态容器又是什么？它是说明Redux有一个专门管理state的地方，就是Store，并且一般情况下是唯一的，应

用中所有state形成的一颗状态树就是Store。Redux由Flux演变而来，但受 Elm 的启发，避开了Flux的复杂性，我们看看其数据流向，如图16-6所示。

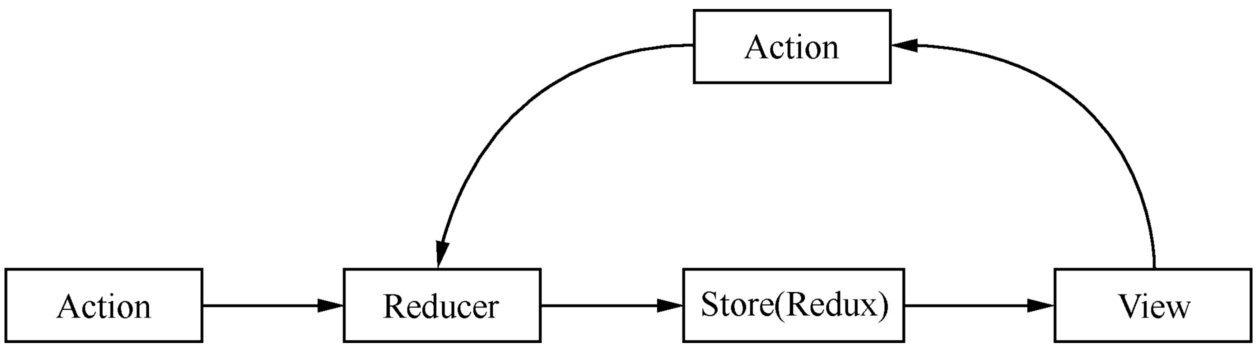


图16-6

不同于Flux架构，Redux中没有dispatcher这个概念，并且Redux设想你永远不会变动你的数据，你应该在reducer中返回新的对象来作为应用的新状态。但是它们都可以用(state, action) => newState来表述其核心思想，所以Redux可以被看成是Flux思想的一种实现，但是在细节上会有一些差异。

如果从MVVM的角度来看，React 是一个纯净的视图库，解决的是组件内部或者继承关系的组件

状态管理。每个组件都没有自己的VM，操作数据需要调用其方法，与jQuery时代别无二致。但是有了Redux，它将所有组件的state组合起来管理，同时指挥这么多组件的变化，其实相当于一个页面级的巨大VM。于是基于此理论，也有人设计出基于反应式的Redux——mobx，如图16-7所示。

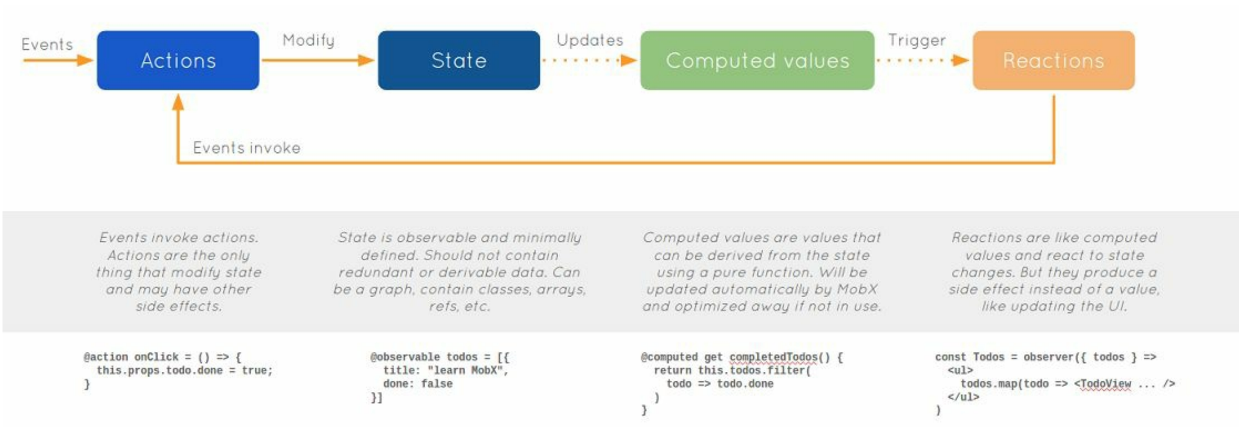


图16-7

16.3.4　React组件的分类

React由于太特立独行，并且由于它能解决移动开发的痛点，于是发展出自己一套完整的生态圈。就像深海中的鲸落，带给我们无数的惊喜。想

理解它们，以旧有的术语来描述它们，只能带给学习者一些错误的印象。因此最后奉上一些有趣的划分，以飨读者。

在地表之上，万物生长依靠太阳。但哪怕是最清澈的海水，在200米以下也几乎是漆黑一片。没有阳光，驱动生物界运行的最主要的能量来源断绝，但是并非没有其他途径。深海海底的生物可以依靠化能合成和海面输送来的物质，热泉口是它们的城市，洋流是它们的道路，从海面缓慢飘下来的食物碎屑（“海洋雪”）是它们的天降甘霖，而偶然落下的巨大身躯，则是它们在大洋荒漠之中的孤岛和绿洲。这些躯体是鲸的尸体，被称为“鲸落”（Whale fall）。

1．容器组件和展示组件

名字来源于redux作者Dan Abramov的博客《Presentational and Container Components》。

如果你将组件分成两类，你会发现它们容易更被重用和理解，这两类称之为容器组件 和展示组件 。笔者也听过其他说法，比如"臃肿的"和"苗条的"，"智能的"和"单调的"，"多状态变量"和"单纯的"，"封装物"和"元件"等。概念不完全一致，却有一样的中心思想。

展示组件 的特点如下。

- 只关注于如何展示。
- 可能同时包含子级容器组件和展示组件，一般含DOM标签和自定的样式。
- 通常用this.props.children来包含其他组件。
- 不依赖app其它组件，比如flux的actions和stores。
- 不会定义数据如何读取，如何改变。
- 只通过this.props接受数据和回调函数。
- 很少有自己的状态变量，即使有，也是UI的状

态变量，比如toggleMenuOpen、Input Focus。
- 除非他们需要的自己的状态，生命周期，或性能优化才会被写为功能组件。
- 例子有Page、Sidebar、Story、UserInfo、List。

容器组件 的特点如下。

- 只关心它们的运作方式。
- 可能同时包含子级容器组件和展示组件，但大都不含DOM标签，而含他们自己所用的wrapping div，从不用自己的样式。
- 为展示组件或其他组件提供数据和方法。
- 调用Flux的actions，并且将其作为展示组件的回调函数。维持许多状态变量，通常充当一个数据源。
- 通常由高阶组件生成，比如Redux里的connect()，Relay里的createContainer()，Flux Utils里的Container.create()，而非手工写出

（注，可能在meteor中数据是例外吧）。

- 例子有UserPage、FollowersSidebar、StoryContainer、FollowedUserList。

2. 智能组件与木偶组件

名字来源于《使用React重构百度新闻webapp前端》，可能也是容器组件与展示组件 的一种叫法。

```
http://wangfupeng.coding.me/share/2016/08/06/restruct-bdnews-webapp-by-react.html
```

原文从如何设计一个新闻webapp引申这些概念。首先将系统划分成若干个页面，然后将每个页面都划分成若干个组件，还要抽象出多个页面中都会用到的通用组件。根据这些组件的颗粒度与对数据的占有情况划分为两种。

智能组件：它是数据的所有者，它拥有数据，

且拥有操作数据的action，但是它不实现任何具体功能。它会将数据和操作action传递给子组件，让子组件来完成UI或者功能。这就是智能组件，也就是项目中的各个页面。

木偶组件：它就是一个工具，不拥有任何数据、及操作数据的action，给它什么数据它就显示什么数据，给它什么方法，它就调用什么方法，比较傻。这就是木偶组件，即项目中的各个组件，如图16-8所示。

16.4　前端路由

将一个页面划分多个切换卡（子页面），然后根据地址栏进行切换，这就是前端路由的由来。前端路由是上面所说的容器组件，是目前我们实现SAP的重要技术之一。Backbone、angular、avalon、vue等都有自己的路由组件。

以avalon组路由为例，分为3部分。

- storage：保存当前的路址 。
- mmHistory：监听地址栏变化或回退按钮，得到当前感兴趣的路径，放到mmRouter里进行匹配。
- mmRouter：定义与解析路由规则，及绑定对应的回调。

这里有几个术语。

路址，是指地址栏中的hash（在JavaScript语言里称url改变该部分不会影响页面重新加载的部分为hash，在后台语言里称之为fragment）或pathname（域名后面的部分）。这是客观存在的地址栏的一部分，用于匹配路由规则。路址必须以/开头，以便在hash模式下区分真正的锚点。

路由规则：地址的抽象形式，一个地址必然匹配某种套路。比如 `https://segmentfault.com/a/1190000004862711`，匹配`/a/:d` 规则，`http://localhost/index.html#/topic` 匹配`/topic` 规则。路由规则也必须以/开头，至少大多数路由器都遵循这个设计。几乎所有路由器，都会将路由规则转换为一个正则。然后正则与它对应的回调组成一个对象，进入一个数组中。当地址过来，就进行遍历，命中了就执行回调！这个方案自后端路由器诞生以来，都是屡试不爽。并且将路由

规则转换为正则都有已成的库了。

```
https://github.com/pillarjs/path-to-regexp
```

像avalon早期也试过使用前缀树方式匹配地址，但支持的路由规则形式太少了，最终也放弃了。

```
https://github.com/RubyLouvre/mmRouter/blob/0.2/avalon.router.js
```

路由参数：路由规则有林林种种的定义方式，但无不例外，它们都支持规则有一部分是可变的。不存在变动部分的叫静态路由，否则称动态路由，大部分路由规则都是动态路由。动态的那部分称之为路由参数。现在流行两种定义路由参数的风格，一是使用`{}`，二是使用:。如`/blog/{name}`或'/blog/:name'。基本上路由参数都位于/的后面，这是因为大多数地址都是遵循RESTful Web API风

格设计的。Backbone就是使用冒号风格，而angular ui-router与avalon路由是两种风格都支持。

路由参数也分3种类型。

必用参数：换言之，可变部分不能省略。如`/blog/{name}`或`/blog/:name`。

可选参数：如果地址栏少了可变部分的，也能认为匹配，这时可能使用默认参数项，通过在后面加？或*来标识。这些符号都是来自正则的元字符，方便转换。如`/blog/{name?}`或`/blog/:name?`

正则参数：当你的路由参数以花括号风格定义，它看起来像一个对象，这时可能用冒号隔开，前面是参数名，后面是一个正则。要求动态也必须符合这个正则才能匹配。如`/blog/{name: ^\d+$}`。

16.4.1　storage

这个用来保持地址栏，防止页面不小心被刷新，导致路由器无法还原当前页面。由于保持的东西非常明确，就是一段字符串，因此它非常精简。基本上用浏览器的两种对象来实现，优先考虑localStorage，其次是cookie。如果想大而全的方案，可以参考下面的连接。

```
https://github.com/nbubna/store
function supportLocalStorage() {
    try {//看是否支持localStorage
        localStorage.setItem("avalon", 1)
        localStorage.removeItem("avalon")
        return true
    } catch (e) {
        return false
    }
}
function escapeCookie(value) {
    return String(value).replace(/[,;"\\=\s%]/g, function (char
acter) {
        return encodeURIComponent(character)
    });
}
var ret = {}
if (supportLocalStorage()) {
    ret.getLastPath = function () {
        return localStorage.getItem('msLastPath')
    }
    var cookieID
    ret.setLastPath = function (path) {
        if (cookieID) {
```

```
            clearTimeout(cookieID)
            cookieID = null
        }
        localStorage.setItem("msLastPath", path)
        cookieID = setTimeout(function () {//模拟过期时间
            localStorage.removItem("msLastPath")
        }, 1000 * 60 * 60 * 24)
    }
} else {

    ret.getLastPath = function () {
        return getCookie.getItem('msLastPath')
    }
    ret.setLastPath = function (path) {
        setCookie('msLastPath', path)
    }
    function setCookie(key, value) {
        var date = new Date()//将date设置为1天以后的时间
        date.setTime(date.getTime() + 1000 * 60 * 60 * 24)
        document.cookie = escapeCookie(key) + '=' + escapeCooki
e(value) + ';expires=' + date.toGMTString()
    }
    function getCookie(name) {
        var m = String(document.cookie).match(new RegExp('(?:^|
 )' + name + '(?:(?:= ([^;]*))|;|$)')) || ["", ""]
        return decodeURIComponent(m[1])
    }
}

module.exports = ret
```

16.4.2　mmHistory

mmHistory的目的是实现window.history的功能。在浏览器中，history对象包含用户已经浏览的URL信息，这就是我们传说中的历史记录。我们可

以通过点击前进后退按钮，重复观看之前的页面。并且history对象也有forward/back两个方法，以编程手段切换页面。当SPA流行后，要在一个页面模拟多个页面的效果，也需要用到history。

我们在上节提到，mmHistory是用来监听地址栏被改动的。其实它也能自动修改地址栏，但要求修改后，页面不能被整体刷新。目前而言，只有两种方式可以实现：修改location.hash与通过pushState、replaceState API。修改地址栏的操作主要用于用户点击A标签时，我们会劫持页面上的所有点击事件，然后判定其事件源，如果是A标签并且符合一系列要求，我们就会手动地址栏，并执行对应的回调操作。

因此mmHistory有3大任务。

- 监听地址栏变化。
- 劫持点击事件。

- 主动修改地址栏。

监听地址栏变化根据浏览器的支持情况也分3种。

- 使用window.onpopstateg事件进行监听history。当用户调用了history.go和history.back方法，或用户按浏览器历史前进后退按钮，都会触发popstate事件。事件发生时浏览器会从history中取出URL和对应的state对象替换当前的URL和history.state。通过event.state也可以获取history.state。

需要注意的是调用history.pushState()或history.replaceState()不会触发popstate事件。因此当我们主动用pushState修改地址栏时，需要手动window.onpopstate() 触发回调。

`pushState()` 与`replaceState()` 的差别是，它

们通常对应旧的`location.assign()`、`location.replace()`，如图16-9所示。

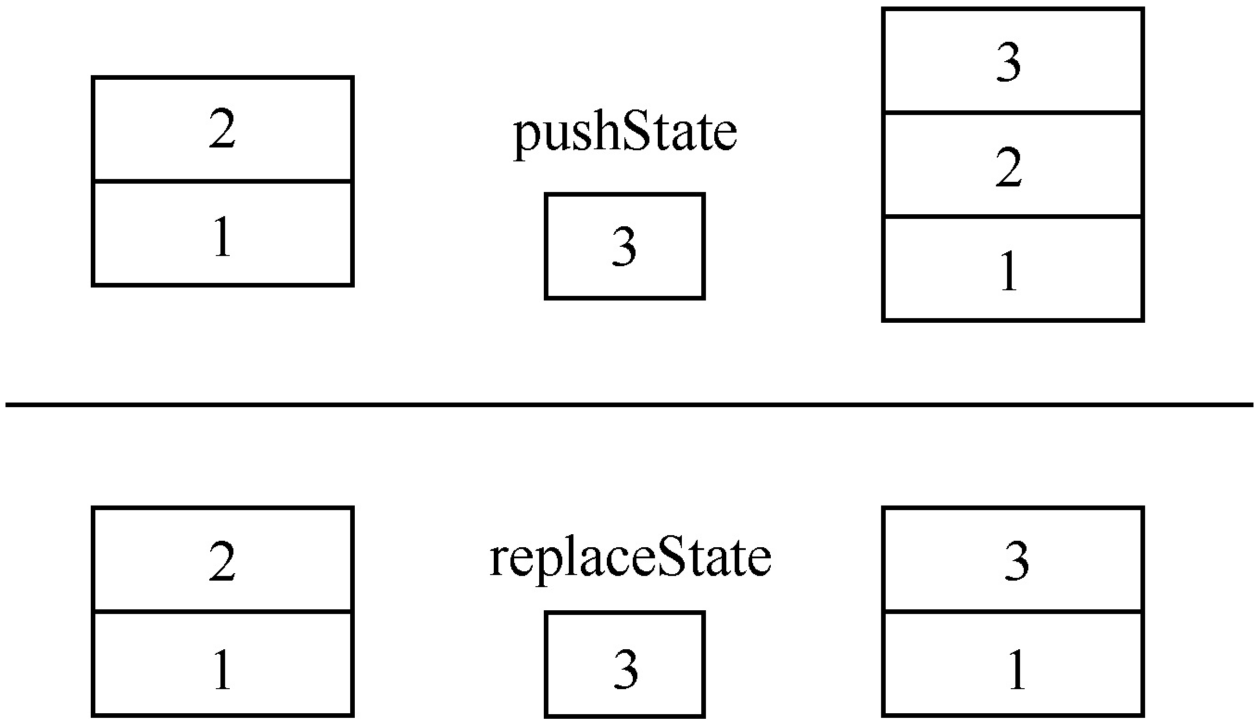


图16-9

- 使用onhashchange事件监听URL的hash变化，这个是从IE8开始支持。当用户使用location. hash = xxx或location.href = xxxx时，只要改动的地方只发生在＃后面就不会引发整页刷新，此事件就会执行。同时，history.go和history.back方法

也会引发此事件。

- IE6、IE7中使用onpropertychange监听document.location属性变动。其他浏览器可以使用interval轮询。至于如何产生历史，就使用iframe hack，这个在jQuery.fn.hashchange等库中已经总结出来了。

下面我们解读一下源码，里面有许多链接，集中了前人的心血。mmHistory基本参考backbone.History，主要方法有start、stop、setHash、onHashChange。先看start。

```
var mmHistory = {
    hash: getHash(location.href),
    start: function (options) {
        if (this.started)
            throw new Error('avalon.history has already been st
arted')
        this.started = true
        //监听模式
        if (typeof options === 'boolean') {
            options = {
                html5: options
            }
        }
        options = avalon.mix({}, defaults, options || {})
        var rootPath = options.root
        var html5Mode = options.html5
```

```
        this.options = options
        this.mode = html5Mode ? "popstate" : "hashchange"
        if (!supportPushState) {
            if (html5Mode) {
                avalon.warn("浏览器不支持HTML5 pushState，平稳退化到onhashchange!")
            }
            this.mode = "hashchange"
        }
        if (!supportHashChange) {
            this.mode = "iframepoll"
        }
        avalon.log('avalon run mmHistory in the ', this.mode, ' mode')
        // 支持popstate 就监听popstate
        // 支持hashchange 就监听hashchange(IE8、IE9、FF3)
        // 否则的话只能每隔一段时间进行检测了(IE6、IE7)
        switch (this.mode) {
            case "popstate" :
                // 此事件在古老的chrome事件有bug，会在页面加载后就自动触发，我们需要延迟绑定此事件 http://code.google.com/p/chromium/issues/detail?id=63040
                setTimeout(function () {
                    window.onpopstate = mmHistory.onHashChanged
                }, 500)
                break
            case "hashchange":
                window.onhashchange = mmHistory.onHashChanged
                break
            case "iframepoll":
            //也有人这样使用 http://www.cnblogs.com/meteoric_cry/archive/2011/01/11/1933164.html
                avalon.ready(function () {
                    var iframe = document.createElement('iframe')
                    iframe.id = options.iframeID
                    iframe.style.display = 'none'
                    document.body.appendChild(iframe)
                    mmHistory.iframe = iframe
                    mmHistory.writeFrame('')
                    if (avalon.msie) {
                        function onPropertyChange() {
                            if (event.propertyName === 'location') {
                                mmHistory.check()
```

```
                        }
                    }
                    document.attachEvent('onpropertychange'
, onPropertyChange)
                    mmHistory.onPropertyChange = onProperty
Change
                }

                mmHistory.intervalID = window.setInterval(f
unction () {
                    mmHistory.check()
                }, options.interval)

            })
            break
        }
        //页面加载时触发onHashChanged
        this.onHashChanged()
    },
    stop: function () {
      //…
    },
    setHash: function (s, replace) {
     //…
    },
    writeFrame: function (s) {
      // …
    },
    syncHash: function () {
     // …
    },
    getPath: function () {
     //…
    },
    onHashChanged: function (hash, onClick) {
    //…
    }
}
```

轮询模式下，需要通过check方法来调用

onHashChanged()，目的只是做一个小小的检测。

```
mmHistory. check = function () {
    var h = getHash(location.href)
    if (h !== this.hash) {
        this.hash = h
        this.onHashChanged()
    }
 }
```

getHash方法是对原生location.hash的不信任搞出来的方法，这里实现太多兼容性问题了。

```
function getHash(path) {
    // IE6直接用location.hash取hash，可能会取少一部分内容
    // 比如 http://www.cnblogs.com/rubylouvre#stream/xxxxx?lang=
zh_c
    // IE6 => location.hash = #stream/xxxxx
    // 其他浏览器 => location.hash = #stream/xxxxx?lang=zh_c
    // Firefox 会自作多情对hash进行decodeURIComponent
    // 又比如 http://www.cnblogs.com/rubylouvre/#!/home/q={%22th
edate%22:%2220121010~
    // 20121010%22}
    // Firefox 15 => #!/home/q={"thedate":"20121010~20121010"}
    // 其他浏览器 => #!/home/q={%22thedate%22:%2220121010~2012101
0%22}
    var index = path.indexOf("#")
    if (index === -1) {
        return ''
    }
    return decodeURI(path.slice(index))
}
```

onHashChanged是用于与mmRouter打交道的桥梁。它存在两种模式，一种是点击模式，被页面的A标签被点击，那么就会将其href 属性加工一下，传进此方法。另一种是事件模式，这时它的第一个参数是事件对象或没有对象（IE6、IE7），需要从地址栏中获取hash。

```
mmHistory.onHashChanged = function (hash, clickMode) {
        if (!clickMode) {
            hash = mmHistory.mode === 'popstate' ? mmHistory.ge
tPath() :
                    location.href.replace(/.*#!?/, '')
        }
        hash = decodeURIComponent(hash)
        hash = hash.charAt(0) === '/' ? hash : '/' + hash
        if (hash !== mmHistory.hash) {
               mmHistory.hash = hash

            if (avalon.router) {//即mmRouter
                hash = avalon.router.navigate(hash, 0)
            }

            if (clickMode) {
                mmHistory.setHash(hash)
            }
            if (clickMode && mmHistory.options.autoScroll) {
                autoScroll(hash.slice(1))
            }
        }
    }
```

由于点击模式下，事件的默认行为被阻止了，因此我们需要手动修改地址栏（也就是里面的setHash方法），此外，有时我们需要将页面滚动条下滑到与锚点同名的元素里，也是在这个方法里做。

```
setHash: function (s, replace) {
     switch (this.mode) {
         case 'iframepoll':
             if (replace) {
                 var iframe = this.iframe
                 if (iframe) {
//contentWindow 兼容各个浏览器，可取得子窗口的 window 对象。
//contentDocument Firefox 支持，> IE8 的IE支持。可取得子窗口的 docum
ent 对象。
                     iframe.contentWindow._hash = s
                 }
             } else {
                 this.writeFrame(s)
             }
             break
         case 'popstate':
             var path = (this.options.root + '/' + s).replace(/
\/+/g, '/')
             var method = replace ? 'replaceState': 'pushState'
             history[method]({}, document.title, path)
             // 手动触发onpopstate event
             this.onHashChanged()
             break
         default:
             //http://stackoverflow.com/questions/9235304/how-t
o-replace-the-location
             //-hash-and-only-keep-the-last-history-entry
             var newHash = this.options.hashPrefix + s
             if (replace && location.hash !== newHash) {
                 history.back()
             }
```

```
            location.hash = newHash
            break
    }
}
```

stop方法比较简单，略过。我们看一下如何劫持所有点击事件，与其内部复杂的过滤条件。笔者是参考了许多库弄成现在的样子。

```
//劫持页面上所有点击事件，如果事件源来自链接或其内部，
//并且它不会跳出本页，并且以"#/"或"#!/"开头，那么触发updateLocation方法
avalon.bind(document, "click", function (e) {
    //https://github.com/asual/jquery-address/blob/master/src/jquery.address.js
    //https://github.com/angular/angular.js/blob/master/src/ng/location.js
    //下面几种情况将阻止进入路由系列
    //1. 路由器没有启动
    if (!mmHistory.started) {
        return
    }
    //2. 不是左键点击或使用组合键
    if (e.ctrlKey || e.metaKey || e.shiftKey || e.which === 2 || e.button === 2) {
        return
    }
    //3. 此事件已经被阻止
    if (e.returnValue === false) {
        return
    }
    //4. 目标元素不A标签,或不在A标签之内
    var el = e.path ? e.path[0] : e.target
    while (el.nodeName !== "A") {
        el = el.parentNode
        if (!el || el.tagName === "BODY") {
            return
        }
```

```
    }
    //5. 没有定义href属性或在hash模式下,只有一个#
    //IE6/IE7直接用getAttribute返回完整路径
    var href = el.getAttribute('href', 2) || el.getAttribute("x
link:href") || ''
    if (href.slice(0, 2) !== '#!') {
        return
    }

    //6. 目标链接是用于下载资源或指向外部
    if (el.hasAttribute('download') || el.getAttribute('rel') =
== 'external')
        return

    //7. 只是邮箱地址
    if (href.indexOf('mailto:') > -1) {
        return
    }
    //8. 目标链接要新开窗口
    if (el.target && el.target !== '_self') {
        return
    }

    e.preventDefault()
    //终于达到目的地
    mmHistory.onHashChanged(href.replace('#!', ''), true)
})
```

最后看autoScroll方法，这主要参考angular 1的同名模块。

```
//得到页面第一个符合条件的A标签
function getFirstAnchor(name) {
    var list = document.getElementsByTagName('A')
    for (var i = 0, el; el = list[i++]; ) {
        if (el.name === name) {
            return el
        }
```

```
    }
}
function getOffset(elem) {
    var position = avalon(elem).css('position'), offset
    if (position !== 'fixed') {
        offset = 0
    } else {
        offset = elem.getBoundingClientRect().bottom
    }

    return offset
}

function autoScroll(hash) {
    //取得页面拥有相同ID的元素
    var elem = document.getElementById(hash)
    if (!elem) {
        //取得页面拥有相同name的A元素
        elem = getFirstAnchor(hash)
    }
    if (elem) {
        elem.scrollIntoView()
        var offset = getOffset(elem)
        if (offset) {
            var elemTop = elem.getBoundingClientRect().top
            window.scrollBy(0, elemTop - offset.top)
        }
    } else {
        window.scrollTo(0, 0)
    }
}
```

最后用angular的$location服务示意图，表示这两种模式（iframepoll也是hashchange的一种补充）的操作原理，如图16-10所示。

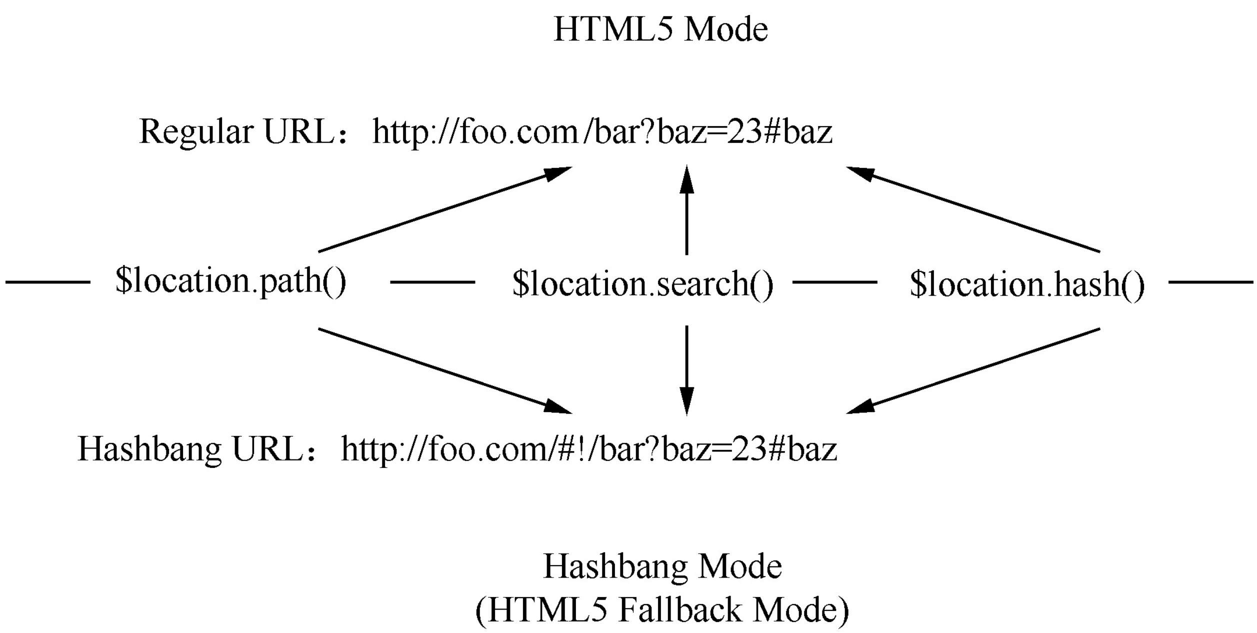


图16-10

16.4.3 mmRouter

mmRouter主要难点是转换路由规则为正则，为简单起见，直接用github上的`path-to-regexp`。

```
var mmHistory = require('./mmHistory')
var storage = require('./storage')
var pathToRegexp = require('path-to-regexp')
function Router() {
    this.rules = []
}
Router.prototype = storage
avalon.mix(storage, {
    error: function (callback) {
        this.errorback = callback
    },
    add: function(){
    },
```

```
    route: function(){
    },
    navigate: function(){
    }
}
module.exports = avalon.router = new Router
```

首先error方法就是添加一个404回调，当一个地址与我们定义好的所有路由规则都不匹配时，就执行它。它相当于React的NotFoundRoute。

接着是添加路由规则的方法，它会转换第一个参数为一个正则。

```
Router.prototype.add = function(path, callback, opts){
      var array = this.rules
      if (path.charAt(0) !== "/") {
          avalon.error("avalon.router.add的第一个参数必须以/开头")
      }
      opts = opts || {}
      opts.callback = callback
      if (path.length > 2 && path.charAt(path.length - 1) === "
/") {
          path = path.slice(0, -1)
          opts.last = "/"
      }
      var keys = []
      var regexp = pathToRegexp('/foo/:bar', keys)
      opts.keys = keys
      opts.regexp = regexp
      avalon.Array.ensure(array, opts)
}
```

route用来判定当前URL与已有状态对象的路由规则是否符合，这是在mmHistory里面调用的重要方法。

```
Router.prototype.route = function (path, query) {
      path = path.trim()
      var rules = this.rules
      for (var i = 0, el; el = rules[i++]; ) {
          var args = path.match(el.regexp)
          if (args) {
              el.query = query || {}
              el.path = path
              el.params = {}
              var keys = el.keys
              args.shift()
              if (keys.length) {
                 _parseArgs(args, el)
              }
              return  el.callback.apply(el, args)
          }
      }
      if (this.errorback) {//404
          this.errorback()
      }
}
function _parseArgs(match, stateObj) {
        var keys = stateObj.keys
        for (var j = 0, jn = keys.length; j < jn; j++) {
            var key = keys[j]
            var value = match[j] || ''
            match[j] = stateObj.params[key.name] = value
        }
    }
```

navigate方法让你手动切换子页面并执行它的方法。

```
Router.prototype.navigate = function (hash, mode) {
     var parsed = parseQuery(hash)
     var newHash = this.route(parsed.path, parsed.query)
     if(isLegalPath(newHash)){
         hash = newHash
     }
     //保存到本地储存或cookie
     avalon.router.setLastPath(hash)
     // 模式0, 不改变URL, 不产生历史实体, 执行回调
     // 模式1, 改变URL, 不产生历史实体,   执行回调
     // 模式2, 改变URL, 产生历史实体,    执行回调
     if (mode === 1) {
         avalon.history.setHash(hash, true)
     } else if (mode === 2) {
         avalon.history.setHash(hash)
     }
     return hash
}
```

基本上这就说完了。

路由器是一个称得上框架 的必备组件，实现各种资源整合的闭环。否则用户引入其他JS生态圈的路由器，会让框架完整性大打折扣。

至此，本书完毕。相对上一版，内容改动较

大，这也说明前端的技术更新迭代太快了。每年都出新的框架与热点技术，这可能是对过去技术的改良，也可能是面临全新的难题应运而生的产物。需要学习的东西太多，根本学不过来。因此必须有自己的方向，不要再妄图“通吃”。有句话说，技术的深度决定广度。很多技术的本源都是相同或相似的，比如说所有面向对象语言，都是基于面向对象思想和原则，比如说设计方案和框架模式，都可以参考前辈们归纳出的各种设计模式。如果精通一门，必势能举一反三，触类旁门其他相似的技术。因此在新的一版中，都会对某项技术进行精讲，然后再给出更多参照用的不同解法，来拓展大家的思路。

最后，重申一下笔者开发框架的三大原则。

（1）复杂即错误 。无论是从开发到维护，都是如此。太过复杂，必须进行分治，进行简化，或

干脆换一种思路。

（2）数据结构优于算法 。参见React的虚拟DOM实现，diff算法很美，但前提是依仗其巧妙设计的虚拟DOM树。为了减轻算法的复杂度，又引进了key属性。

（3）出奇制胜 。已有的生态太强大，做第二个jQuery的意义不大，必须引进颠覆式创新，升维思考，降维打击，改变旧有的格局。

彩蛋

本书每一章开头都有一幅图，代表一种设计模式，它可能与本模块有关。现在是开谜的时候，不知聪明的你猜到没有？

章 节	画 画	含 义
种子模块	遥控器	门面模式
语言模块	桥梁	桥接模式
浏览器嗅探与特征侦测	独一无二的你	单例模式
类工厂	厂房	工厂模式
选择器引擎	思维脑图	策略模式
节点模块	羽毛	享元模式
数据缓存模块	记事本	备忘录模式

样式模块	首饰	装饰器模式
属性模块	中介	代理模式
PC端的事件系统	报纸	订阅发布模式
移动端的事件系统	转换插头	适配器模式
异步模型	一环套一环	责任链模式
数据交互模块	脸谱	原型模式
动画引擎	构建工具webpack	解释器模式
MVVM	水的形态变化	状态模式
组件	乐高	组合模式

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。


异步社区
书架
通知
首页
图书
电子书
文章
写作
新年新气象
HELLO!
社区UI全新改版，崭新面貌迎接2017！为答谢社区用户，
即日起到1月26号
全场电子书8折优惠！
前端开发
数据科学
编程语言
移动开发
游戏开发
机器学习&深度学习
更多>>
免费电子书
Free eBook
立即获取
我要写书
Write for Us
立即查看
Python机器学习——预测分析核心算法
贝叶斯方法：概率编程与贝叶斯推断
机器学习项目开发实战
贝叶斯思维：统计建模的Python学习法
近期活动

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券 。 使用方法：注册成为社区用户，在下单购书时输入“**57AWG** ”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区

微信订阅号

微信服务号

官方微博

QQ群：436746675

社区网址： www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn